NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

USING THE BOOTSTRAP CONCEPT TO BUILD AN ADAPTABLE AND COMPACT SUBVERSION ARTIFICE

by

Lindsey Lack

June 2003

Thesis Advisor:	Cynthia Irvine
Second Reader:	Roger Schell

Approved for public release; distribution is unlimited

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.			
1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** June 2003	**3. REPORT TYPE AND DATES COVERED** Master's Thesis	
4. TITLE AND SUBTITLE: Using the Bootstrap Concept to Build an Adaptable and Compact Subversion Artifice			**5. FUNDING NUMBERS**
6. AUTHOR(S) Lack, Lindsey A.			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000			**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A			**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.			
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release, distribution is unlimited			**12b. DISTRIBUTION CODE**

13. ABSTRACT ***(maximum 200 words)***

The attack of choice for a professional attacker is system subversion: the insertion of a trap door that allows the attacker to bypass an operating system's protection controls. This attack provides significant capabilities and a low risk of detection.

One potential design is a trap door that itself accepts new programming instructions. This allows an attacker to decide the capabilities of the artifice at the time of attack rather than prior to its insertion. Early tiger teams recognized the possibility of this design and compared it to the two-card bootstrap loader used in mainframes, since both exhibit the characteristics of compactness and adaptability.

This thesis demonstrates that it is relatively easy to create a bootstrapped trap door. The demonstrated artifice consists of 6 lines of C code that, when inserted into the Windows XP operating system, accept additional arbitrary code from the attacker, allowing subversion in any manner the attacker chooses.

The threat from subversion is both extremely potent and eminently feasible. Popular risk mitigation strategies that rely on defense-in-depth are ineffective against subversion. This thesis focuses on how the use of the principles of layering, modularity, and information hiding can contribute to high-assurance development methodologies by increasing system comprehensibility.

14. SUBJECT TERMS System Subversion, Computer Security, Artifice, Trap Door, Bootstrap, Assurance, Layering, Information Hiding, Modularity			**15. NUMBER OF PAGES** 90
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UL

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release, distribution is unlimited

USING THE BOOTSTRAP CONCEPT TO BUILD AN ADAPTABLE AND COMPACT SUBVERSION ARTIFICE

Lindsey Lack
Civilian, Naval Postgraduate School
B.S., Stanford University, 1994

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 2003

Author:

Lindsey Lack

Approved by:

Dr. Cynthia E. Irvine
Thesis Advisor

Dr. Roger R. Schell
Second Reader

Dr. Peter J. Denning
Chairman, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

The attack of choice for a professional attacker is system subversion: the insertion of a trap door that allows the attacker to bypass an operating system's protection controls. This attack provides significant capabilities and a low risk of detection.

One potential design is a trap door that itself accepts new programming instructions. This allows an attacker to decide the capabilities of the artifice at the time of attack rather than prior to its insertion. Early tiger teams recognized the possibility of this design and compared it to the two-card bootstrap loader used in mainframes, since both exhibit the characteristics of compactness and adaptability.

This thesis demonstrates that it is relatively easy to create a bootstrapped trap door. The demonstrated artifice consists of 6 lines of C code that, when inserted into the Windows XP operating system, accept additional arbitrary code from the attacker, allowing subversion in any manner the attacker chooses.

The threat from subversion is both extremely potent and eminently feasible. Popular risk mitigation strategies that rely on defense-in-depth are ineffective against subversion. This thesis focuses on how the use of the principles of layering, modularity, and information hiding can contribute to high-assurance development methodologies by increasing system comprehensibility.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

EXECUTIVE SUMMARY

It was said of the far-flung Roman Empire that all roads led to Rome – north, south, east and west. And it was through that road system that Caesar extended his writ far and wide. And they were great roads. But there is a funny thing about roads. They go both ways, and when the Vandals and the Visigoths decided to attack Rome they came right up the roads.†

Like the roads of the Roman Empire, the roads that make up the Information Superhighway go both ways as well. Like the Roman network of roads, the Internet has been responsible for tremendous improvements in efficiency and productivity. However, it also exhibits the Pandora's-box-like quality: every improvement is linked with a danger.

Understandably, the security threats that receive the most attention are those that are observed such as web-site defacements and viruses that arrive with email messages. The threats more deserving of concern are those more likely to have a significant impact—such as a disruption of critical infrastructure or a compromise to national security—and less likely to be observed. For many decades, security researchers have been aware of the most potent threat: operating system subversion. System subversion is the intentional modification of a system prior to its production use (such as during development or distribution), which allows the subverter to completely bypass the system's security controls. The code that achieves this bypassing of controls is known as a "trap door" or an artifice. Because planning is required, subversion is typically seen as a professional attack, one most likely to be conducted by a nation state or other visionary organization.

This thesis builds upon the previous research on subversion that has been conducted at the Naval Postgraduate School. In 1980, Philip Myers published a thesis that outlined the desirable characteristics of a subversion artifice [MYER80]. Myers highlighted six desirable characteristics of a trap door artifice, including compactness, adaptability, possession of a unique trigger, untraceability, and independence from both operating system upgrades and variations in installation.

† From Friedman, Thomas L., *The Lexus and the Olive Tree: Understanding Globalization*, Anchor Books, p 322, May 2000.

In 2002, Emory Anderson demonstrated the ease with which a subversion artifice may be implemented, adding 11 lines of code to Linux that bypassed the Network File System permission controls [ANDE02]. The behavior of the Anderson artifice was static, lacking the characteristic of adaptability mentioned in the Myers thesis.

The idea of an adaptable artifice originated with some of the earliest vulnerability assessment teams (also known as "penetration" or "tiger" teams), who were well aware of trap doors. The teams envisioned that an artifice could be constructed that would take additional instructions as input, thereby allowing the artifice to be programmed. The concept of a programmable artifice was likened to a traditional bootstrap mechanism, wherein the initial input is composed of instructions that enable the system to load even more instructions.

This thesis provides a demonstration of an adaptable artifice, showing that adding the characteristic of adaptability to the artifice requires little additional effort or technical expertise. Bootstrapping is the key to satisfying the inherently contradictory goals of compactness and adaptability. The artifice demonstrated in this thesis – consisting of 6 lines of C code that can be added to the Windows XP operating system – is able to accept additional code, thereby supporting the implementation of an arbitrarily complex subversion, and allowing an attacker to program the artifice at the time of attack rather than having to predict the desired capabilities prior to its insertion. The implemented artifice exhibits all six characteristics identified by Myers as desirable.

The threat from subversion is both extremely potent and eminently feasible. Nevertheless, the dominant risk mitigation strategy, rather than increase the assurance level of the most critical assets, is to use multiple solutions in parallel to achieve "defense-in-depth." This paradigm is so accepted, in fact, that it has become an official part of the Department of Defense's information assurance policy. Unfortunately, the solutions in question, such as firewalls, intrusion detection systems, etc., have all been shown to be ineffective against subversion.

The only accepted solution for ensuring the absence of a subversive artifice is to apply a strict development methodology specifically designed to provide high-assurance. This thesis describes how some of the accepted but frequently overlooked principles of

programming contribute to this methodology. Specifically, the thesis discusses layering, modularity, and information hiding, and how these principles contribute to reducing complexity, achieving system comprehensibility, enabling the application of formal methods, and segmenting a trusted computing base from non-trusted components. The thesis also addresses the impact of the use of highly structured techniques on system performance.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

It was said of the far-flung Roman Empire that all roads led to Rome – north, south, east and west. And it was through that road system that Caesar extended his writ far and wide. And they were great roads. But there is a funny thing about roads. They go both ways, and when the Vandals and the Visigoths decided to attack Rome they came right up the roads.†

We live in the Information Age. Today the important roads are the ones that carry ones and zeros rather than those that carry modern-day chariots. While the technology market bust at the turn of the century reduced the use of hyperbole, the radically transforming nature of the information technology, from microchips to the Internet, cannot be denied, and indeed is hard to overstate.

Yet for all the dramatically positive attributes that information technology-related progress has brought, there is a dark side to the progress as well. While information technology has become very adept at expanding capabilities and features, it has made comparatively little progress at providing security. This is not to deny the prodigious efforts that have been focused on security, especially recently, as the extraordinary growth in the field has exacerbated security issues. Corporations have been developing and refining a multitude of security products, such as firewalls, intrusion detection systems, and system security enhancements. The reason that little true progress has been made is that most current efforts are focused on the wrong part of the problem. Security consists fundamentally of two components: a mechanism that implements some security feature, and assurance that the mechanism operates as advertised. One would not, for example, hire a bodyguard if the bodyguard were not both capable and trustworthy: one attribute without the other is of little use. The problem with the current approaches to security is that efforts focus on the mechanisms. As every new threat or technique arises, a security mechanism or feature is developed to address the threat. Unfortunately, adding one mechanism on top of another mechanism can do nothing to increase the assurance of

† From Friedman, Thomas L., *The Lexus and the Olive Tree: Understanding Globalization*, Anchor Books, p 322, May 2000.

the overall system. Any weakness in the foundation of the system will be exploitable, even if the higher supported layers are strong.

The truly critical issue today is the lack of product diversity. Without a rich variety of systems from which to choose, with each system providing a different level of assurance to meet the needs of different risk profiles, effective management of risk is impossible. Imagine that an IT manager's only choices are between one system that does not meet the assurance requirements but does meet functionality and budgetary requirements, and another system that exceeds assurance requirements but whose functionality is untested and whose cost far exceeds the limitations of the budget. What is the correct choice in this scenario? Obviously there is none.

A couple decades ago, the potential threats to computer systems were understood and research was conducted into how to build computer systems that were capable of enforcing security policies with a high level of assurance. In the mid-80's, the government tried to establish a marketplace for systems with a range of assurance levels; however, for a variety of reasons this marketplace was never firmly established.

Since that time many things have changed: computers have become ubiquitous, and in today's highly networked world, the need for secure computer platforms is higher than ever. Ironically, the operating systems commonly available do not take advantage of previously developed techniques for designing secure systems.

One of the most significant threats against computer systems is the subversion of an operating system: the intentional modification of a system to allow the complete bypassing of security controls via a "trap door." In March of 2002, Emory Anderson published a thesis [ANDE02] demonstrating the implementation of a subversion artifice, showing that relatively little technical expertise is required to carry out such an attack. His demonstration exhibited several advantageous traits, such as compactness (with the accompanying natural resistance to discovery), possession of a unique trigger, and untraceability. His artifice, however was static, being pre-programmed with a specific behavior, and thus giving the impression that it would be difficult for a potential subverter to effectively deploy a subversion attack against a specific target.

The idea that it would be possible to design a subversion artifice that could accept new code was realized at the very beginning of research on computer security. Roger Schell described the idea's origin [SCHE03]:

> During some of my early tiger team participation with Jim Anderson and others, it was recognized that a significant aspect of the problem of Trojan horse and trap door artifices was the ability of the artifice itself to introduce code for execution. A self-contained example was a subverted complier in turn emitting an artifice, as hypothesized in the early 1970's Multics evaluation by Paul Karger and me [KARG02], which stimulated Thompson's discussion of this in his Turing lecture [THOM84]. Soon after Karger's report, other tiger team members observed that the ultimately desired artifice did not have to be self-contained, but could be imported later. It was suggested that a particularly insidious packaging of this could have the initial artifice provide the functions of simple bootstrap loader typically hardwired in the computers of that era. These loaders did something like read the first two binary cards from the card reader and initiate execution of what was read, which was usually a further bootstrap program to read and execute additional binary cards. Hence this class of attack came to be commonly referred as the "2-card loader problem."
>
> The concept and term became quite commonplace, although I don't know of any widely reported actual implementation. Myers during his 1980 research at NPS was well aware of the 2-card loader problem, and his thesis implicitly included this in the trait of a trap door he termed "adaptability" which included being "designed to modify operating system code online." [MYER80]. Much later Don Brinkley and I in our 1995 IEEE essay had the 2-card loader problem in mind when we briefly described a hypothetical attack where, "Among the functions built into the Trojan horse was the ability to accept covert software 'upgrades' to its own program." [BRIN95].

The hypothesis of this thesis is that creating a flexible subversion artifice – one which allows the desired subversion to be programmed at the time of attack – is not much more difficult than creating a fixed artifice, and well within the capability of a professional attacker. Additionally, such an artifice could retain the characteristics that have been demonstrated in static artifices, such as compactness, possession of a unique trigger, and untraceability.

The demonstration associated with this thesis is a collaborative effort. This thesis deals with only one portion of the overall artifice design, specifically the compact toe-

hold code which is inserted into the operating system, and the bootstrapping of that toe-hold into a base or foundation that provides some simple services to the designers of a specific artifice. Two other theses are contributing towards the other aspects of building the entire artifice. David Rogers is working on the methods by which the artifice may be expanded to an arbitrarily large size, dynamically linked, and made persistent [ROGE03]. Jessica Murray is developing a specific artifice that subverts the IPSEC subsystem, causing it to reveal its traffic and keys to the attacker [MURR03]. All three efforts are being supported by Microsoft Corporation, and are using Windows XP as the target of subversion.

The rest of the thesis is organized as follows. Chapter II provides a background on the history of secure systems, classifications of threats, and details on the subversion threat in particular. Chapter III describes the details of the demonstration of an adaptable subversion artifice. Chapter IV provides an analysis of the difficulty involved in implementing the artifice and the impact on the threat model. Chapter V outlines measures that may be taken to counter the subversion threat, and specifically addresses the importance of modularity, information hiding, and layering to the overall measures taken. Chapter VI concludes the thesis.

II. BACKGROUND

A. HISTORY OF SECURE SYSTEMS

In today's increasingly automated and networked world, the issues of security and privacy in relation to computing systems are increasingly becoming mainstream topics. Attacks that occur against the infrastructure of the Internet or major Internet companies are headline news. Today, one might overhear conversations on the street about solutions for firewalling home computer systems, or about a recent run-in with a computer virus spread through email. Some prominent cyber-criminals have achieved infamy through their exploits, and in several cases following their incarceration have become high-paid security consultants. To many the recent phenomena of personal computers, cell phones, and the Internet—markers of the Information Age—seems to be a Brave New World. This new world, in addition to bringing remarkable capabilities and potential for new efficiencies, brings with it in Pandora-box fashion new dangers and threats.

Today's common computer user must deal with a myriad of issues regarding computer security. He must make sure he is using virus protection software and that the software is updated with the latest signatures. He must ensure that security patches for his operating system(s) are administered. He must worry about whether his home network and systems have an adequate firewall. He must maintain his own backup files to insure against any unforeseen loss. Yet how must this person feel about the state of computer security when the parties who are truly concerned with computer security such as prominent Internet-based companies and the government, are themselves vulnerable to cyber-attack. Although most of the press revolves around low-level attacks such as web-defacement, it is not surprising that more significant attacks fail to reach the media due to the potential damaging publicity that could impact both corporate and government interests. It may surprise today's average computer user to know that most of the computer security issues we face today, including the most potent threats, were well understood three decades ago. What may be even more surprising is that the research conducted at that time, at the start of the Information Age, actually resulted in the

production of systems that were significantly superior to today's systems with regard to security, but that for a variety of reasons failed commercially.

The first computers were tremendously expensive devices: the computer's time was many magnitudes more valuable than the operator's time. Since the computer worked so much more quickly than the operator, the obvious solution was to program the computers to service many operators at the same time. This time-sharing feature unfortunately created a new computer security threat. Whereas before time-sharing the access to the computer could be controlled with physical security, once time-sharing was developed, physical security could no longer be used. With time-sharing, the critical difference was that the security mechanism of the computer system itself was programmed, opening up the possibility that someone could reprogram (and thus subvert) the security mechanism. This observation was first made in the Air Force Computer Technology Planning Study [ANDE72]. This report introduced the concept of a security kernel, one core part of the system that integrates all security functions, and the concept of a Reference Monitor, an abstraction that serves to formalize the notion of protection. The primary purpose of these concepts is to make the security components small enough that they can be analyzed and understood completely, thereby providing assurance that the desired protection principles are satisfied and that no subversion of the protection mechanism has been conducted.

The first commercially available system designed around the security kernel concept was the SCOMP [FRAI83]. However, not all operating systems adopted these principles. The Multics operating system, for example, failed to incorporate this design despite the fact that it was specifically designed to meet requirements for security. In 1974 Karger and Schell published an evaluation of the security of the Multics operating system. In this evaluation they showed that although Multics was designed with security goals in mind and assumed to be secure by its designers and users (and indeed was significantly more secure than common contemporary systems), it was still extremely vulnerable to the introduction of malicious software.

In 1973, Bell and LaPadula published a report [BELL73] showing that it is possible to mathematically formalize the specifications of a security policy that specifies

what it means to preserve confidentiality. Later, Biba [BIBA77] showed that a similar model could be used to describe information integrity. These are known as mandatory policies.. A mandatory access control policy is a homomorphism on an access control matrix, creating specific equivalence classes of subjects and objects and reasoning about those classes. Mandatory policies only work, of course, if the classes are permanent and may not be reassigned (or reassignment would take place outside of the model). Note that the policies that are to be enforced, be they discretionary or mandatory, are orthogonal to system enforcing the policy. The security-kernel approach suggested in the Anderson report recommended keeping the security-kernel minimized so that it could be verified as correctly enforcing the policy.

The policy is important to the design of secure system as well, however, due to the likely avenue of attack. While one potential tactic is to attack the security mechanism of the operating system itself, another is to try to masquerade as a user with a higher level of privilege. Since the privilege levels of users, including the administrator, are typically encoded within the policy, mandatory access control policies can help ensure that attempts to gain additional rights are restricted. Especially useful in this regard is the Biba policy, which can serve to maintain the integrity of components of the operating system.

In 1985, the government published criteria for evaluating the assurance class of an operating system, called the Department of Defense Trusted Computer Evaluation Criteria (TCSEC) [DOD85]. With the support of the government guidelines created in the TCSEC, many commercial attempts were made at producing a secure operating system to support the needs of the government. Although some systems were created and successfully evaluated, for a variety of reasons these attempts were not in the end a market success.

Today more than ever there is a need for high-assurance operating systems. Today's de facto standard for operating systems severely limits choices. Given that different users of information technology have dramatically different tolerances for risk, the limitations dictated by the few dominant commercial operating systems have the

potential to result in potentially catastrophic outcomes. Loscocco highlights some of the needs that are directly addressable today [LOSO98]:

- Within a Public Key Infrastructure, the need to maintain highly secure Certificate Authority sites, as well as the need for the protection of end-systems with regard to the integrity and confidentiality of public and private keys (respectively) and the integrity of the public-key process.
- For encrypted connections, the need to maintain the security of the end-points and in some cases a local key management server (for IPSec).
- For voting applications, the need to maintain the integrity of the voting mechanisms of the end-user voting terminals.

As pointed out by Loscocco, the creation of any secure application without a secure operating system on which to run the application is tantamount to building a "Fortress built upon sand." That is, if the operating system is weaker than the application (or related cryptographic operation), then the operating system will become the target of an attacker. Once the operating system is compromised, the secure application is invariably compromised as well.

B. THREATS TO INFORMATION SYSTEMS

In 1980, Myers outlined a taxonomy for categorizing the ways in which internal computer system controls may be attacked. He divided potential attacks into three categories, inadvertent disclosure, penetration, and subversion. Inadvertent disclosure relies upon a confluence of events (which may include human error) that conspire to reveal information that should have been kept confidential. Penetration describes a situation in which the attacker takes advantage of a flaw in an operating system to gain unauthorized access or control. The skill level of the attacker is not specified: it could be someone who is inexperienced and may be easily caught or a professional cracker. Subversion is the intentional undermining of the computer systems security controls through the introduction of an artifice, or piece of code, at any point in the system lifecycle. In Myers' ontology, subversion includes both trap doors and Trojan horses. Trap doors are distinguished by the fact that they may be activated directly, whereas a

Trojan horse lures a user to activate it, and is limited to acting on behalf of a user. The following figure shows this categorization:

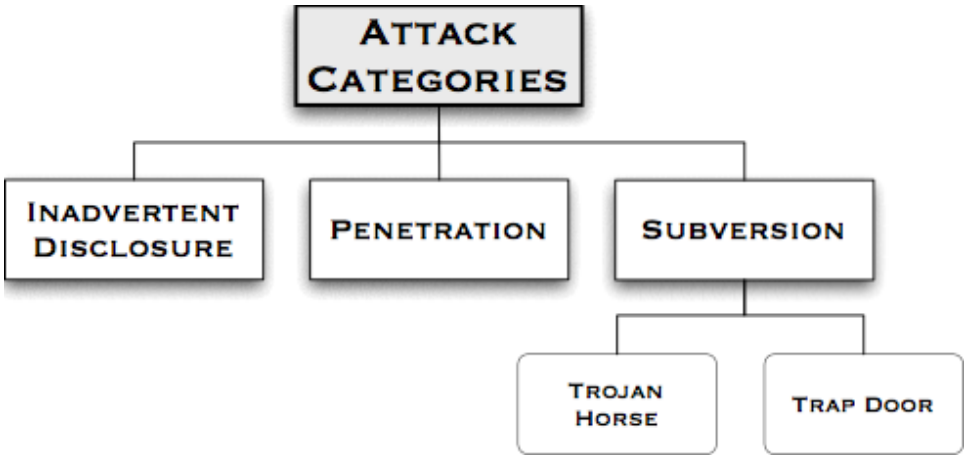

Figure 1. Myers' Attack Categories

Much has changed in the last 23 years since Myers published his thesis, yet the categorization he proposed is still relevant. In 1980, the term "script-kiddie" had not been invented, whereas today it sees common use. Likewise, today people use the term "spyware" to refer to software that, in addition to its primary capabilities that are marketed, covertly sends information about the user to a centralized site to ostensibly assist with the effective targeting of advertisements. Today, a trap door is likely better known as a "root kit," software that is installed after a penetration takes place that allows the attacker to regain access to the compromised system while evading detection. The fact that the most common method for installing a trap door is to first penetrate a system illustrates the magnitude of the problem today. Indeed, the ease with which today's systems may be penetrated may be part of the reason why subversion garners so little attention. If penetrating a system were not so trivial, attackers would begin to look at targeting a system earlier in its lifecycle.

Myers' categorization includes a characterization of the attacker. Myers maintains that although the technical skills required for any of the specific attacks is not very great, by definition the act of subversion implies a more disciplined and more methodical

approach to the attack, one which would be carried out by a professional or group of professionals with a specific objective. Penetration, in contrast, is painted as the type of activity that is today attributed to script-kiddies: actions by a lone user with limited aims.

Today, the mapping between the seriousness or sophistication of an attack and the attacker's technical expertise is less clear. For example, imagine that a company that provides free software for playing music has also added code in the software that scans one's computer for cookies and keywords that indicate the interests of the user, allowing the company to provide targeted advertising. Such Trojan horse code certainly violates the security policy that most users would like to enforce, in this case referring to the personal privacy component of the policy. Although such an example may be characterized as subversion since the implementers were certainly skilled, motivated, and well organized, it likely does not match the profile of a serious, professional subverter. Also, software tools have been developed that make implementing a Trojan horse trivial. Often such techniques are used simply to gather passwords in order to further the penetration exploits of non-serious actors. Likewise, as Myers readily indicates, an attacker utilizing penetration is not necessarily an amateur. Some of the tools that have been developed to conduct penetration are in fact very sophisticated, and penetrations by skilled attackers are often accompanied by the installation of root kits to establish a back door.

Despite the inconsistency of the models that has arisen due to the sheer magnitude of the progress that has been made in information technology, the intent of the taxonomy is clear and remains valid: with regard to computer system attacks, there is a tremendous difference between non-serious and serious actors. Non-serious actors typically act alone or in loose-knit groups, use tools that others create, and have limited goals, motivation, and time-frames. Serious actors are highly-skilled and highly-organized, operate as an organization rather than as individuals, have aggressive goals and strong motivation, and are capable of long-term planning. Non-serious actors will tend to employ techniques that are convenient, whereas serious actors will employ techniques that best enable them to achieve their purpose.

The one technique that is solely in the toolkit of a serious actor – and would be unlikely to be used by a non-serious actor – is the insertion of a trapdoor subversion artifice into a system early in its lifecycle (before deployment). Note that this does not include a root kit installed after a penetration has occurred. The property that makes this particular technique relevant to only the serious actor is the timeframe and planning involved. A non-serious actor would be unlikely to have a long enough planning horizon to make such efforts meaningful, and would also be unlikely to have the organizational resources needed to carry out such an effort. A trapdoor artifice has a couple important advantages over a Trojan horse. First, a Trojan horse requires the cooperation of the victim (although the victim may be unaware of having cooperated), thereby reducing the attacker's ability to plan and to be certain of outcomes. Second, a Trojan horse is limited to operating within the privilege level of the user. Although crackers have become very proficient at escalating privilege levels on common operating systems, a system that can effectively separate privilege levels, such as one enforcing a mandatory access control policy, makes this attack of questionable value. A trap door, on the other hand, is activated by the attacker directly, and is essentially unlimited (that is, limited only by design) in how it can subvert the protection mechanisms of the operating system. Although a trap door is designed to circumvent normal system controls, the circumvention process does not necessarily draw attention to itself since the artifice can be designed to be a part of the system: technically there is no distinction between the artifice and the operating system; the artifice that subverts the protection mechanism is the operating system.

Although there are some additional hurdles to overcome in order to install a trapdoor artifice, there are some compelling advantages. Subversion in general has the advantage, as already mentioned, of being unlikely to be detected. This is because the subversion is designed to operate either as the operating system or as a user (in the case of a Trojan horse), and therefore its actions are not considered to be "abnormal." A trap door subversion will ideally be controlled by an activation mechanism. When it is not needed, it is deactivated, making detection of its existence much more difficult.

Unlike a root kit that is installed following a penetration, a subversion artifice does not leave a trail of evidence, digital bread crumbs if you will, that point to its existence. A penetration will necessarily cause a change to the system that can be tracked. Even a very skilled attacker is unlikely to be able to completely erase all traces of a penetration. File integrity verification tools (such as Tripwire), the built-in auditing features of a file system that track all accesses and modifications to files, the traces of files that remain on a disk even when files are overwritten, and even the memory that is overwritten when a buffer overflow takes place all potentially contribute to the trail of evidence that may be forensically examined to verify that a penetration has taken place and to explain how it was performed. With subversion, such a trail of evidence does not exist, so it is much more difficult to know whether your system has been subverted or not.

Another advantage of a trap door subversion is that it is possible to deploy a subversion artifice widely, even if the subversion will only be used in a limited context.

C. SUBVERSION VIA TRAP DOOR ARTIFICE

1. Desirable Artifice Traits

Myers details several of the traits that are considered to be desirable in an artifice. These include compactness, revision independence, installation independence, untraceability, being uniquely triggerable, and adaptability. I will explain each of these in turn, clarifying why these traits would be useful, and how the subversion example provided by Anderson measures up to the desired traits.

a. Compactness

Compactness contributes to the inherent ability of the subversion artifice to remain undetected. Since the only way to determine whether an artifice exists is to understand the actual code being carried out by the system, maintaining compactness makes positively identifying subversion code much more difficult. Anderson was able to implement his demonstration of subversion with only 11 lines of C code.

b. Possession of a Unique Trigger

He followed the prescription Myers gave for the trap door precisely, adding artifice code to recognize the activation or deactivation trigger, and based on that trigger set or disable the master mode, which in his case affected the file access permission checking code.

c. Revision Independence

Revision independence is tied to the nature of the attacker, a serious actor with a long planning horizon. Such an attacker might install the artifice early in the lifecycle of the system, and may not know exactly when the artifice will be used. If the artifice is corrupted by later revisions, the investment made by the attacker will be lost. Therefore, it is important to ensure that the artifice will survive system updates and revisions. Myers suggests that device drivers are a good choice since they generally don't change with software revisions, and since they are often coded in low level languages they are more difficult to understand, and therefore less likely to be identified as subversive. Myers, however, perhaps did not anticipate the tremendous pace of change that has taken place in the world of hardware in the past two decades. This rapid pace of change in hardware has actually reversed the trends one may have reasonably expected about hardware and software.

The software with the highest rate of change has been the software closely associated with hardware, and software with the lowest rate of change has been the more abstract "higher level" code that implements standard protocols. For example, many systems today still carry the original TCP/IP code that helped to make TCP/IP a worldwide standard. In contrast, the entire architecture of Windows device drivers has been redesigned several times, and each new architecture has generally maintained little compatibility with previous architectures. Naturally, each new architecture has required completely new code for each individual device. Also, hardware components themselves have very short production lifecycles. Even software that is part of the operating system, but that interfaces with device drivers, is subject to fairly frequent modification, as shown by the fact that the Windows Network Device Interface Specification (NDIS) has undergone a major revision with each new operating system update.

Given these facts, it appears that for an artifice to achieve revision independence, it is better to insert the artifice in code that provides compatibility with widely-adopted standards. The Anderson demonstration (in part) is installed in the software that implements the TCP/IP stack, which would be likely to survive operating system revisions.

d. Installation Independence

Installation independence is important when the same operating system may contain different components depending upon the needs of the particular installation. This applies to custom-designed solutions that may only include specific components in a particular installation, a practice that was likely more common a couple of decades ago. However, it also applies to the installation approach common in today's systems, which is to install a standard operating system platform that includes a superset of what most customers need, and turn off the components that are not desired. For example, if a particular installation does not provide network access to a file system for remote access by clients, then an artifice that bypasses the permission checking of only this service will not be usable. Thus, the Anderson demonstration arguably does not provide a high degree of installation independence.

e. Untraceability

Untraceability is achieved by minimizing the impact on the system, which can take the form of either an audit trail or changes to the system that can be examined forensically. A trap door subversion has the advantage of operating as the system itself. Contrariwise, a Trojan Horse, since it operates effectively as the user who ran the code, is subject to the auditing procedures that apply to all users. The Anderson demonstration exhibits a high degree of untraceability.

f. Adaptability

Adaptability allows a subversion artifice that is planted at one point in time to be modified at some point after the insertion. The Anderson demonstration exhibits little adaptability, since the artifice has few options. Although system files with new artifices may be loaded onto the system using the artifice, the artifice itself has limited state. The logical extreme of adaptability is complete programmability.

2. The Erosion of Concern about the Subversion Threat

The Myers thesis, which highlighted the subversion threat, contributed to the development of criteria that could be used to ensure the development of systems that are verifiably secure, e.g. that can be shown to be free of a subversion artifice. For nearly two decades, efforts were made to address the subversion threat. The government, in addition to creating the TCSEC, mandated that for high-assurance needs the government only use systems certified as meeting the requirements of the TCSEC. This prompted development efforts by commercial companies to develop systems to meet the high-assurance requirements. For a variety of reasons, including the fact that government agencies failed to heed the mandate, the efforts to create a marketplace for systems that could provide a rich variety of assurance levels failed. Today, the TCSEC has been replaced by the Common Criteria, and although companies who produce COTS products do continue to achieve certification of their existing products at the medium assurance levels, the marketplace for high assurance systems, including those capable of addressing the threat of subversion, has dried up. Speculation regarding the social and political factors that produced an initial surge of corporate development and the following cancellation of such efforts is beyond the scope of this thesis.

The reaction to the failure of high-assurance efforts has been curious, however. It appears as though the impression that no adequate solution exists to address the threat of subversion has given rise to the opinion that subversion is no longer a significant threat. Given the importance of information technology to the future and the fact that these perceptions are driving critical policy decisions, the magnitude of the danger that such misjudgments represent is enormous.

The turning point that officially marked the government's capitulation on its efforts to create high-assurance systems was the signing of DoD Directive 8500.1 October of 2002 [DOD02]. This new directive cancelled DoD Directive 5200.28, "Security Requirements for Automated Information Systems (AISs)," [DOD88] which had enforced the application of the TCSEC to automated systems. Whereas the philosophy of the former security requirements focused primarily on assurance measures—ensuring that systems are designed from the ground up to meet specific

assurance objectives—the philosophy of the new directive is focused on achieving security goals through the application of "Defense in Depth." The directive defines "Defense in Depth" as:

> The DoD approach for establishing an adequate IA posture in a shared-risk environment that allows for shared mitigation through: the integration of people, technology, and operations; the layering of IA solutions within and among IT assets; and, the selection of IA solutions based on their relative level of robustness.

Robustness, as defined by the document, is intended to address assurance issues, albeit with more flexibility and less precision than the previous directive. The key change with the defense in depth approach is that there is now the idea that one's information assurance posture may be raised via the layering of information assurance solutions. This principle of "security through addition" is illustrated by such measures as active penetration testing, perimeter defense, and intrusion detection. Unfortunately, history has shown that all of these measures fail against the threat of subversion, even when several measures are used in parallel. The perception that additional measures lead to additional mitigation of risk is false. Evidence of the inadequacy of testing and perimeter defenses are provided in the Emory Anderson thesis [ANDE02].

Prior to DoD Directive 8500.1, a report on the insider threat [OASD00], published in 2000 by the Office of the Assistant Secretary of Defense, illustrated the perception that subversion is not a significant risk. This report downplays the subversion risk by claiming that it is too hard. It posits that an individual developer of a COTS product intent on subverting the system would have "an extraordinarily difficult task" in targeting an individual customer since the production and distribution of the product is independent from the developer. The would-be subverter "would have to deliver the same product to all customers while retaining the ability to isolate a particular customer for exploitation."

That such a subversion is not "extraordinarily difficult," but is rather exceedingly easy is precisely the objective of this thesis. The fact that the author, in a short amount of time, was able to create a subversion artifice that could be inserted into every copy of a

product, yet be selectively activated and customized, shows the technical feasibility of such a task, and hopefully will serve to highlight the danger of underestimating the threat of subversion.

THIS PAGE INTENTIONALLY LEFT BLANK

III. DEMONSTRATION OF AN ADAPTABLE SUBVERSION ARTIFICE

A. PURPOSE OF DEMONSTRATION

1. High Level Goals

The purpose of this demonstration is to show that it is possible, and indeed relatively easy, to design and insert a compact and adaptable subversion artifice into an operating system. The key consideration here is that these two characteristics are typically mutually exclusive: adding flexibility to software generally results in bloated code rather than compact code. This demonstration focuses on the construction of the mechanism whereby a very compact artifice can be dynamically grown, essentially the "2-card loader" envisioned by many early security tiger teams. The end product is an artifice foundation or base that does not itself provide much functionality. Instead it provides a platform that may be used to implement a subversion of arbitrary complexity and sophistication.

Although the actual demonstration is implemented on a single platform, the concepts behind the design of the subversion are platform independent and could be implemented on any common operating system that has not been designed using high assurance methodologies.

While it is valuable to understand how easy or difficult detection of an artifice is, obfuscation is specifically not an objective of this demonstration. There are numerous ways to try to actively hide the existence of an artifice. One of the points of this thesis is that, even without the attempt to disguise an artifice, it is still nearly impossible to detect due to its inherent characteristics.

2. Assumptions

The assumption of this demonstration is that the subversion in question is being carried out by an insider: someone with access to the code who is responsible for legitimate portions of the code. The real risks are not, however, limited in by this assumption. Whereas the demonstration assumes knowledge of the source code and the

ability to insert modified code at a specific point in the development lifecycle, Myers points out that risks are present at every stage of the system lifecycle, including design, development, distribution, maintenance, etc. While having access to the internals of the system is an advantage, there are many examples of malicious software that have found a way to make use of undocumented aspects of system internals. These examples include kernel-level root kits, which can produce results similar to this demonstration, as well as exploits for discovered vulnerabilities. These examples do show that having source code is not necessary (and sometimes not even sufficient!) for understanding the workings of the system.

There are several technical assumptions that were made in the design of this demonstration:

- The target for the subversion is based upon Windows NT technology, which means that the actual target may be Windows NT, Windows 2000, Windows XP, or Windows XP Embedded.
- The subversion will make use of network connectivity, and the target will have a standard networking mechanism.
- The memory of the target should be sufficiently unconstrained to allow for the storage of small amounts of assembly code. This does not assume that storage is infinite, but that a relatively small and straightforward program should be supportable.
- The system can be made to allow the modification of memory that contains other kernel executables.
- The target system will use the Intel x86 machine instruction set. Note that this is not a design limitation: machine code for the loaded artifice could be designed to meet the requirements of any chip.

3. Requirements

Several requirements arise out of the definition of an effective subversion mechanism by Myers, as described above:

- Compactness
- Possession of a unique trigger
- Untraceability
- Revision and installation independence
- Adaptability

The primary requirements for this demonstration are the characteristics of compactness, possession of a unique trigger, and adaptability. In addition to the requirements outlined by Myers, the author added several additional requirements to increase usability and to support the fact that the artifice is programmable. These include:

- Integrity verification
- Feedback
- Session tracking
- Hard reset

a. Compactness

The property of compactness can be indicated roughly by the number of lines of code added to the system. The subversion artifice demonstrated in the Anderson thesis was 11 lines of code; this demonstration must be within the same order of magnitude.

b. Possession of a Unique Trigger

The possession of a unique trigger is required since the user needs to be able to turn the artifice on and off, and it is desired that such activation and de-activation not be likely to be performed by accident.

c. Untraceability

The desire that the artifice be untraceable is related to the desire of the attacker to avoid detection. While a flexible artifice might be programmed to take actions that could be traced, the fixed component of the artifice should avoid such actions.

d. Revision and Installation Independence

In order to maximize the effective lifetime of the artifice within the system, it should be placed in a location that is unlikely to be revised. Likewise, to ensure the highest probability of being effective at the target site, it should be placed in a location that has a high probability of being a part of the target installation.

e. Adapatability

Since this demonstration produces only the base artifice, the artifice should provide some amount of primary memory for the use of holding code that allows the execution of that code. This is the fundamental measure of adaptability, that the artifice builder may expand upon the base element to create an arbitrarily large and sophisticated subversion. The amount of memory provided for the user should be sufficiently large to enable arbitrary expansion. For example, in order for the user to build a larger artifice elsewhere in memory, a certain amount of bootstrap code is necessary to find that area and load the necessary code. The artifice should at a minimum provide sufficient space for such code. In practice, this means that there is enough free memory in the base element to hold the code that discovers or requests additional memory and manages that additional memory.

f. Integrity Verification

Given that the information coming into the system will likely be coming through an imperfect channel, it is desirable that the integrity of the content be verified in order to increase the usability of the artifice and to reduce the chance of inadvertent failure of the target system. Integrity checking also contributes to the ability of the artifice to avoid detection, since a failure condition would potentially provide the victim with an obvious signal that something is amiss.

g. Feedback

In order to improve the usability of the artifice, the artifice base should support communication back to the subverter. The subverter should be able to customize and program the content of this communication, but the artifice base should supply this service to its customers. Note that this requirement is not necessary for a subversion effort to succeed. It merely provides a level of usability to the attacker, which is particularly useful for demonstration purposes, both in order to produce an illustrative demonstration and to support debugging

h. Session Tracking

Also, to improve the usability of the system if there are going to be several attackers working in concert, the system should provide a service that keeps track of who is using the system in order to reduce conflicts. The session tracking may be used to ensure that only one user uses the artifice at a time. Although the session ID must be known by the attacker, it is not otherwise intended to support authentication.

i. Hard Reset

For reasons related to the preservation of integrity and the tracking of user sessions, the artifice should have hard reset functionality in the case of an emergency. This ensures that if the artifice enters a state that appears problematic, such as being unreachable or not displaying the expected behavior, it can be reset to its original state to ensure accessibility

B. DESIGN DECISIONS

In order to conduct the bootstrap transition, from a “toe-hold” to an artifice that creates the desired subversion of the operating system, it is necessary to transmit the new code into the system. The options available for establishing a communication channel are limited only by the imagination. Virtually any input mechanism may be used, such as the keyboard, the network, serial ports, etc. Messages could be hidden in files that are read by the system, such as documents, graphics files, or even music files, and these files could enter the system via a variety of methods, including email, web downloads, FTP file transfers, peer-to-peer file sharing, or even via the physical transfer of removable media. For this demonstration, the network has been chosen for simplicity as well as for

its increasing ubiquity. It should be noted that although two-way traffic is assumed to be possible for convenience sake, the design of the artifice does not rely on the use of outbound traffic. Given that the initial bootstrap portion of the artifice is a part of the networking code, it was decided that the base portion of any artifice should include networking services that may be used in the design of more sophisticated artifices. There were a couple of guiding principles that drove the design of the base portion of the artifice:

- The base element of the artifice should be involved with any use of the network that is programmed into more complex elements of the artifice.
- Of the base element of the artifice, the "toe-hold" should be as compact as possible.

In order to keep the subversion artifice compact, yet also provide services that allow one to easily develop a custom subversion, a bootstrap-like mechanism has been adopted. The primary purpose of the first part of the bootstrap mechanism is simply to provide a "toe-hold," a means by which something may be inserted into the system. This analogous to the traditional 2-card loader of the mainframe era, in which the first two punch cards inserted contained the code necessary to assist in loading the subsequent cards. The purpose of the second stage is to assist the building of the third stage. The third and final stage of the base artifice expands the set of services that are provided to someone constructing a custom-build artifice.

This expansion incorporates generic mechanisms that would be used by anyone wanting to load their own artifice. This bootstrap mechanism confers adaptability due to the fact that only the toehold must be determined at the time of insertion. While the base artifice gives the attacker the ability to load and execute arbitrary code, the second and third stages of the artifice are themselves adaptable. If, for example, the attacker determines that feedback from the subverted system is not important, but that another feature is important for the base artifice (or perhaps that there is not even a need for the features of the base artifice), such changes can be made after the toe hold is inserted and just before the modified code is shipped to the system through the toe hold.

The inspiration for the design borrows from the techniques that have been developed by the "blackhat" community. Techniques for exploiting buffer overflow errors are widely available, and in some cases have grown fairly sophisticated. Buffer overflow exploits take advantage of the fact that arbitrary data can be placed into memory and then executed. The design of the subversion in its simplest form replicates this behavior. While the subversion artifice will be explicit, it should be noted that a similar effect may be achieved by intentionally creating code that is vulnerable to a buffer overflow. It is important to point out the differences between the explicitly coded artifice and a buffer overflow vulnerability. A buffer overflow vulnerability is by its nature less noticeable than explicit lines of code since there are no lines of code directly associated with the vulnerability: the vulnerability is created by the absence of sufficient error checking. This is not to say that just because an artifice has specific lines of code associated with it that it is obvious, however, given the amount of code that exists in a system and the difficulty of comprehending the function of each line of code.

On the other hand, a buffer overflow may be observed fairly easily even without source code by observing the behavior of the system. A subversion artifice is generally implemented with a trigger mechanism, so that only when a specific input key or signature is delivered will the artifice take action. Therefore, testing for such a trigger via trial and error will be nearly impossible. Additionally, although not addressed by this thesis, it is conceivable that an explicit artifice trigger could be obscured in the same fashion as a buffer overflow (not represented by individual lines of code) while retaining possession of a trigger key. One other advantage of the explicit artifice is that it tends to be more user-friendly. It can be designed to provide a user with predictable behavior and not impact the system it is on, whereas exploiting a true buffer overflow usually means damaging some part of the running state, which is often more difficult to recover from cleanly.

1. TCP/IP Versus NDIS

In deciding exactly where to insert the artifice into the networking modules of the operating system, the initial inclination was to use the very low layers that are close to device drivers. In Windows systems, there is a layer called the Network Device Interface

Specification (NDIS), which is designed to provide a standardized layer between the device drivers that power the networking hardware and the operating system. The initial reasoning for this decision was that it would result in an artifice that could operate in a protocol independent manner, yet not be dependent upon the particular hardware used, as a device driver would be. Although inserting an artifice in a device driver itself has some implementation advantages, such as generally less oversight and testing and ease of introduction into the kernel due to an overly-trusting driver model, it was not considered due to the more important goal of reaching installation independence.

The final decision was to use the TCP/IP stack rather than the NDIS layer. Although NDIS is protocol independent, the use of TCP/IP is so prevalent that the advantage of protocol independence is negligible. Also, the TCP/IP stack is more static that the NDIS layer. Whereas the NDIS version number has changed with virtually every new operating system version released, the TCP/IP functionality has remained relatively static. Thus, using TCP/IP contributes to the goal of revision independence. The overriding factor, however, was based upon the desire to keep the artifice both compact and usable. Putting the code within NDIS would have required the addition of some kind of checksum within the artifice to verify the integrity of incoming information. Since the incoming traffic would be code (due to the need to grow the artifice), unverified data could have severe consequences, likely crashing and losing control of the target. By using a UDP packet, the artifice is able to leverage the checksum performed to know that the packet has not been corrupted in transit. An additional feature of the UDP checksum operation is that a packet with a bad checksum is silently dropped, and a specific bad checksum could potentially be used as a trigger. This has the effect of being able to communicate with the artifice in a way that is resistant to both testing and auditing.

2. Network Stack Visibility Considerations

Hiding its existence was not an explicit goal of the artifice. Nonetheless, questions that seek to understand whether the artifice would really be found are certainly of interest and pertinent to the topic. Placing the artifice in the networking stack was largely done because it was easy: since the decision was made to transmit the artifice over the

network, subverting the networking code was a natural decision. There are other questions that get more to the point:

- Even if the victim knew that the artifice was in the networking code, would that really substantially change the difficulty of finding the artifice?
- How difficult would it be to obscure the artifice, making it appear as though it belongs in the code?
- Would it be possible to achieve the same effect (compromising the network stack) without putting the toe-hold there?

One of the arguments for why the artifice "toe-hold" is difficult to detect depends upon size. The artifice is very small, whereas the amount of other code in the system is very large. While the code involved with networking is smaller than the code for the entire system, there is still a substantial amount of code. The ratio of the number of lines of code in the networking system to the number of lines of code associated with the toe-hold is still massive.

Additionally, no attempt was made to disguise the intent of the artifice code. There are likely many potential ways of doing so, such as following the naming conventions of the surrounding code and using variable names that might be confused with valid variables. Additionally, creating an artifice toe-hold by failing to validate input, for example, would make the artifice much more difficult to detect (similar to replicating a buffer overflow, as previously discussed).

A similar effect as the toe-hold code could certainly be achieved by modifying code in another (non-networking) part of the system. For example, if the location of packets could be known or perhaps stored, it would be possible to use a frequently occurring system event to trigger a check of all packets in memory. Examples of frequently occurring functions include clock-based functions, disk access, and object manipulation. Such a design would rely on the fact that messages sent to the victim machine would not have to be reliable. If a given packet were not observed, another could be sent. In fact that attacker could repeat the necessary messages until they were all observed.

C. DETAILED DESCRIPTION OF ARTIFICE

The structure of the base element of the subversion was broken into three stages, the "toe-hold" portion of the artifice, which is the initial phase, the loading of the artifice base, which is the second phase, and the completion of the construction of the artifice base, which is the third stage. Each stage provides a different set of capabilities.

1. Stage 1 – Toe-hold

The first stage of the artifice bootstrap (shown in Figure 2) is known as the "toe-hold" code. The term bootstrapping refers to the act of "pulling oneself up by one's bootstraps." Although the imagery of the term suggests the impossible, in reality there is always some initial support needed in order to start a bootstrap, i.e. the toe-hold. In this demonstration, this is the code that is inserted into the source code sometime during the product lifecycle prior to the start of the attack. The toe-hold code provides three primary capabilities. First, it provides the memory that will be used by the artifice to store additional code and data. The amount of primary memory is sufficient to implement a small artifice or allow the construction of a larger artifice. Second, it supports a hard reset of the subversion mechanism back to a known initial state. This is done by loading a second stage that simply performs a "return" instruction. Third, it is able to detect and act upon specific signatures occurring in networking traffic sent by the subverter for the purpose of loading the code for the second stage or jumping to the start of the code for the second stage. Stage 2 code must fit in a single packet.

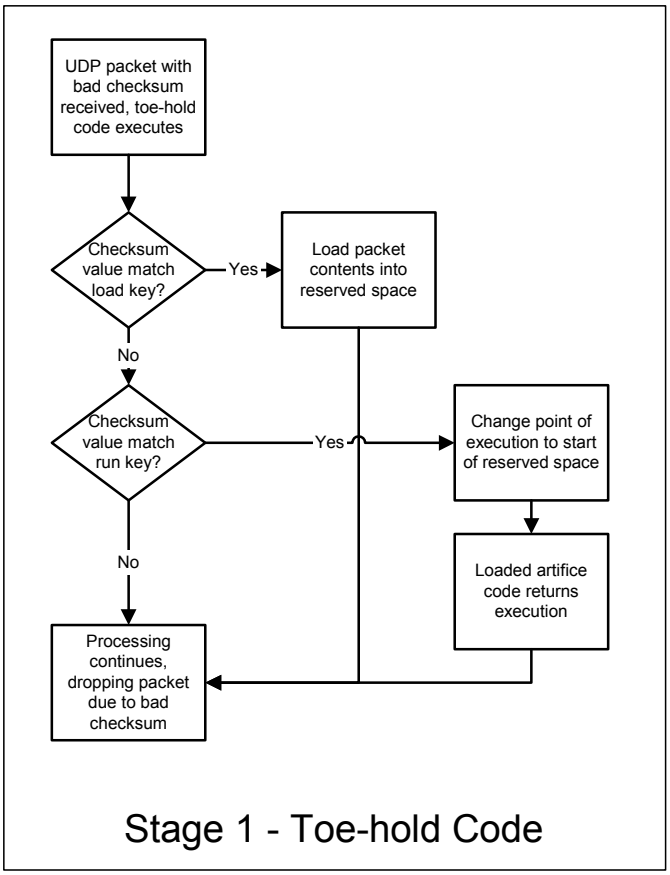

Figure 2. Stage 1 Logical Flow

The artifice toe-hold code consists of several small elements that are added to the UDP code. These include:

- A global array that provides limited storage for the artifice base and for a small amount of additional subversion code.
- A unique load key that triggers the loading of the packet contents into the code array.
- A unique start key that triggers the point of execution to jump to the code in the code array.
- A small amount of logic that occurs after the UDP code has established that the packet contains a bad checksum value. This logic, based upon the existence of one of the two unique keys, either loads the packet contents

into the start of the array or starts executing the code at the start of the array.

The implemented toe-hold code consists of 6 lines of executable C code added to the UDP-handling routines. The six executable lines of code implement the logic that checks for the two unique keys, and either loads the contents into the array or calls the array as though it were a function. In addition to the 6 lines, there are two lines used for declarations: one declares an array that reserves the primary buffer space for the artifice, and the second is a type definition that allows the pointer to the array to be treated as a function.

The purpose of the toe-hold code is to load and start the second stage of the bootstrap code. This is analogous to a traditional hardwired bootstrap loader. Whereas the historical 2-card loader required that the bootstrap code be stored in the first two cards, this bootstrapped artifice requires that the second stage code be contained in a single packet. The two unique keys that identify a packet as being intended for the artifice could of course coincide with a packet that has legitimately computed these same checksums. Indeed, if one assumes that any given packet will randomly generate a 16-bit checksum, then it will take only an average of 16,384 UDP packets before one of the two triggers is duplicated. There are two mitigating factors, however. First, after artifice code is loaded, any packet indicating that the artifice code should be executed will be checked by the artifice code to ensure that the correct session key is present in the incoming packet. If the 16-bit session key provided in the packet does not match the current session key, no action is taken. Second, a packet is checked for the particular unique checksum only when the checksum fails. Assuming that packets are corrupted in transit rarely, let us say only one in 10,000 packets and that UDP traffic represents some fraction of traffic observed such as 25%, that means that an accidental activation of the artifice to load the contents of the packet would occur on average about once every 1,310,000,000 packets. If the attacker desired to have less chance of an accidental reset of the loaded artifice, she could certainly modify the toehold code to include a check on a later segment of the incoming packet as is done in the checking for the session key.

The toe-hold code is the only "hard-coded" portion of the artifice. All further steps are completely flexible. Although, for example, Stage 2 may be implemented to accept exactly 4 packets in order to load Stage 3, that implementation is arbitrary and can be replaced if the design of Stage 3 changes. If, for example, the subverter has a very simple exploit that does not require the services of the artifice base, the exploit may be loaded directly into the primary buffer. On the other hand, if more features are desired for the artifice base code, Stage 2 may be written to load Stage 3 with 8 packets.

Because the toe-hold is hard-coded, it is the component that must provide the hard-reset functionality. In order for the subverter to conduct a hard reset, she merely has to send a packet with the unique load key and contents that indicate that the execution, if it enters the primary buffer, should simply return.

In addition to restoring the system to a known state, the subverter may also want to cover any traces of activity. If the only evidence of activity is the initial portion of the primary buffer that may be rewritten with the contents of a single packet, the subverter may only need to send the one hard reset packet to cover all evidence of activity. Should the subverter wish to overwrite the entire primary buffer, she may need to send three packets: one containing code that will overwrite the entire buffer as desired, a second to execute that code, and a third to overwrite the just executed code. Of course, if the subverter has modified areas of the system beyond the primary buffer, he would need to clean up those other areas prior to overwriting the primary buffer.

2. Stage 2 – Loading of Artifice Base

The second stage of the bootstrap process (shown in Figure 3) consists of the code loaded by the first stage. The purpose of the second stage code is to load the last stage of the artifice base.

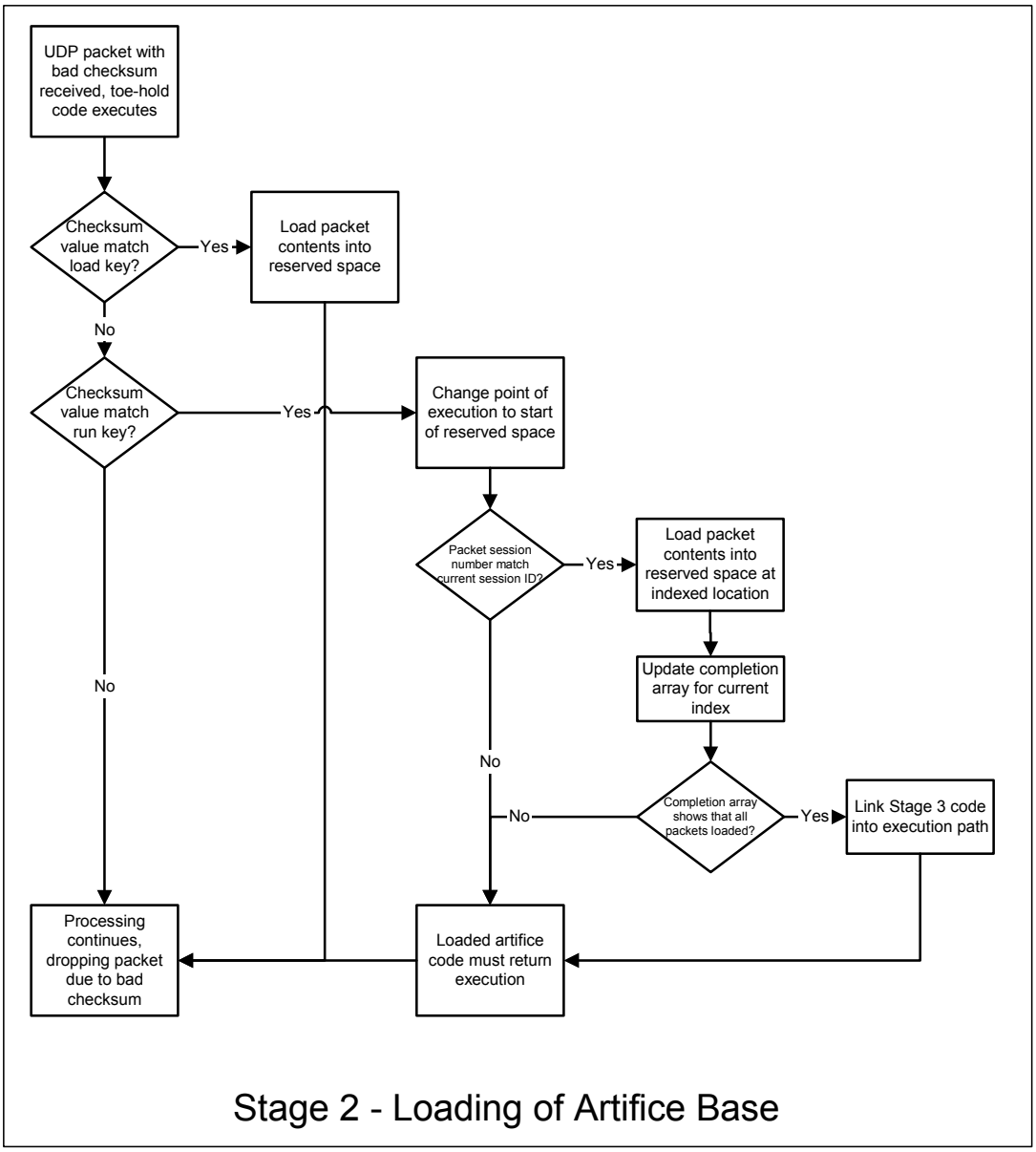

Figure 3. Stage 2 Logical Flow

Whereas the code for the second stage of the artifice base must fit within a single packet, the code for the third stage may require multiple packets. Thus the second stage of the bootstrap process must manage the loading of multiple packets. In order to do this, the code keeps track of which portions of Stage 3 code have been loaded and which ones remain. One additional feature of the second stage is that it allows the subverter who initiates the loading of the second stage to use a unique session key, so that attempts by

multiple subverters to interact with the artifice concurrently will not lead to system failure.

The second-stage code consists of a couple of elements that facilitate the loading of the third stage, including:

- A variable that identifies the current session.
- An array of bits that indicate whether the associated part of Stage three code has yet been loaded.
- Logic that ensures that the incoming packet matches the current session.
- Logic that identifies which part of the Stage three code the current packet represents, loads the contents of the packet into the correct location, and registers that the portion in question has been loaded.
- Logic that checks to see whether the loading of the third stage is complete, and if it is, it sets a new session ID, and modifies the second stage code so that the next packet will be processed by the Stage three code.

The actual code for the second stage is similar to the "shellcode" or "asmcode" that is used in buffer overflow exploits. It is passed into the system as raw opcodes, and is designed to be position independent since the actual location of the code may be unpredictable.

The third stage will be loaded by the second stage using a fixed number of packets, each with a code payload of a fixed size. Thus, the packets may arrive in any order. If any packets fail to arrive, they may be safely sent again until all the packets are successfully received.

The inclusion of Stage 2 is optional, depending upon: the size of the Stage 3 code, the amount of code that the subverter wishes to send in each packet, and the maximum amount of data that may be sent in a packet. Because the incoming packets are one of the few indications to the victim that subversion is taking place, the subverter may elect to restrict himself to smaller packets in order to reduce the likelihood of detection (assuming

the subverter believes that more numerous smaller packets are less obvious than a single larger packet).

3. Stage 3 – Completion of Artifice Base

The third stage of the bootstrap (shown in Figure 4) begins once the code delivered to the second stage has been completely loaded. This code is known as the "artifice base" and provides three main features. First, it supports the loading of additional code into a limited area of memory. Second, it supports the assignment of multiple triggers to different points of execution, so that specific code may be activated as desired via the network. Third, it provides the capability for feedback back to the subverter via the network to let the subverter know the status of the artifice and any additional information that the artifice may be programmed to provide.

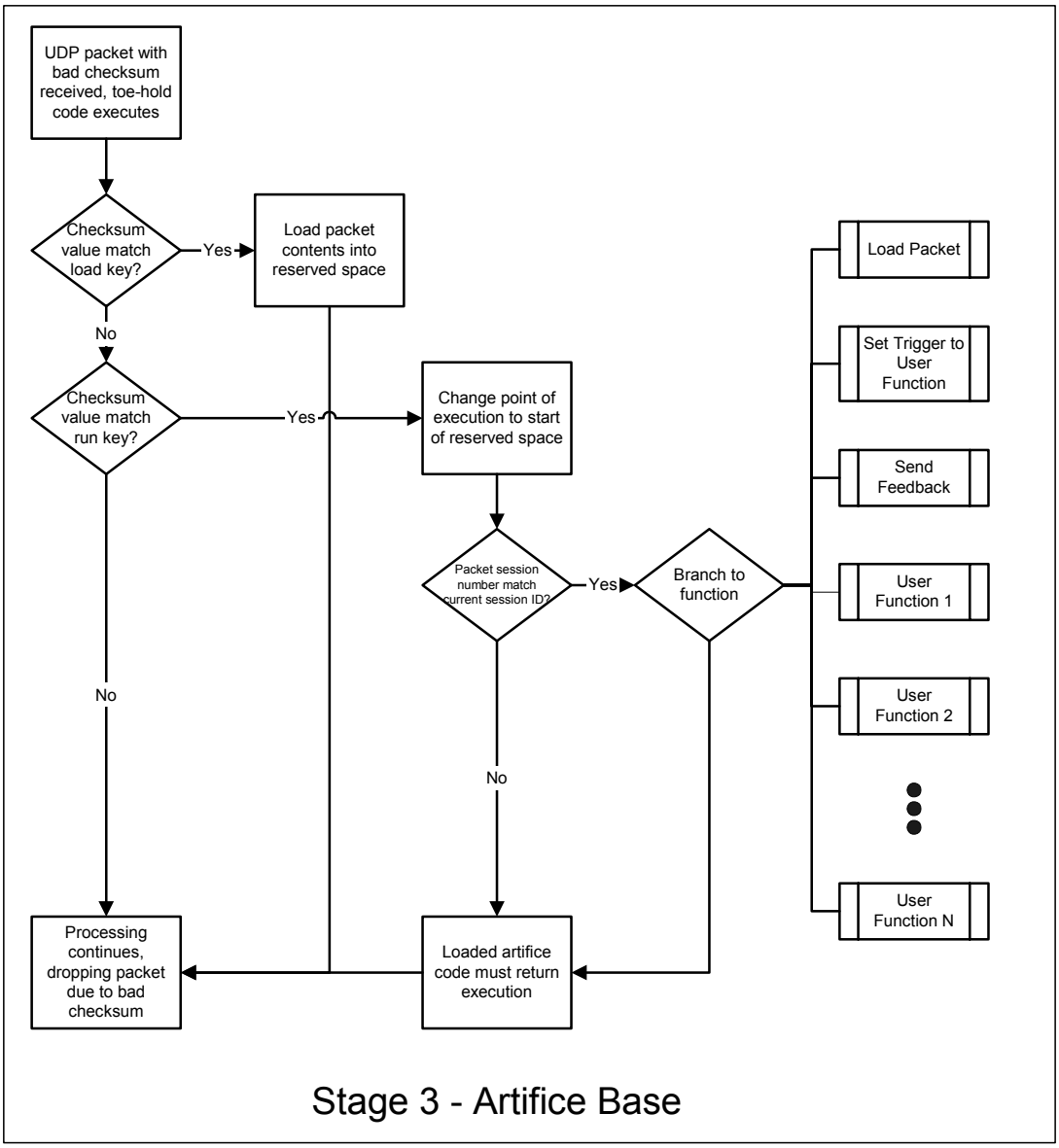

Figure 4. Stage 3 Logical Flow

Like the second stage code, the third stage code is composed of raw opcode instructions, and is position independent.

The artifice base supports four basic functions: loading of code, setting of code triggers, executing code, and providing feedback. The loading function allows code sent to the artifice to be copied to a specified location within the primary buffer space. The trigger setting function allows a numbered trigger to be associated with a specific section of code. Also, before associating a trigger number with the starting address of the loaded

code, the function uses a checksum to verify the integrity of the loaded block of code. The function to execute code simply passes the point of execution to the correct location based upon the specified trigger. The function to provide feedback may be triggered independently, or it may be specified as an option in conjunction with any other command. Each of these functions may be "called" by sending an appropriately formatted packet to the artifice. Using this foundation, the potential subverter may construct his own arbitrary set of functions.

4. Packet Interface Specifications

a. Hard-Coded Load Packet

Figure 5 below shows the format of the UDP header that should be used to load the artifice buffer with code. This is used in Stage 1 to load Stage 2 code, and to conduct a hard reset.

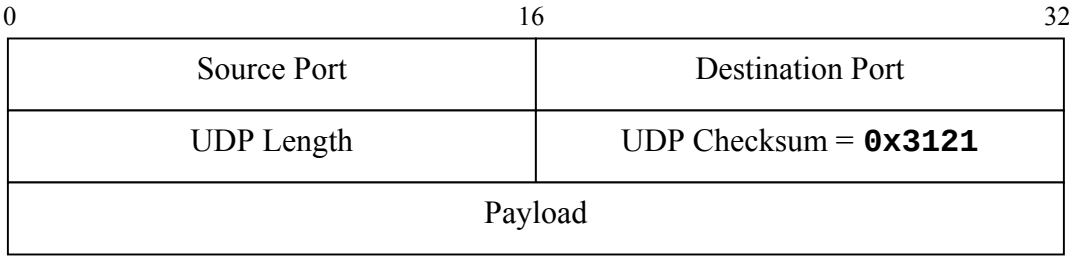

Figure 5. UDP Header Specification for Load Packet

When the UDP checksum field contains the value of `0x3121` (and that value is an incorrect checksum for the given packet), then the payload of the UDP packet is loaded directly into the primary receive buffer. In the rare situation where `0x3121` is actually the correct checksum for the packet, the user would have to modify the packet in some way in order to get the payload to load correctly. The entire payload is copied into the buffer, as indicated by the UDP Length. No error checking occurs with regard to the length: since the primary buffer allocated by the initial artifice code is designed to be larger than the maximum payload size, critical data should not be overwritten.

b. Hard-coded Start Packet

The following diagram shows the format of the UDP header that should be used to jump the point of execution into the primary buffer. This format is used for every packet sent to the artifice once code has been loaded into the primary buffer. In later stages, the UDP header remains the same, while the payload of the UDP buffer is formatted in accordance with the particular artifice code being executed.

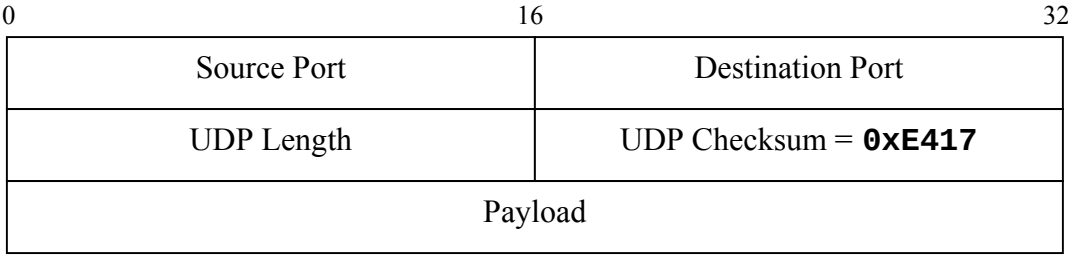

Figure 6. UDP Header Specification for Start Packet

When the UDP checksum field contains the value of `0xE417` (and that value is an incorrect checksum for the given packet), then a function call is made to the address of the start of the primary buffer. Although the primary buffer is declared as an array, the address of the array is copied into a variable that represents a function pointer. The manufactured function receives a single parameter of type `void*`, which is used to pass the starting address of the UDP packet into the artifice code in the primary buffer. This allows the artifice code within the primary buffer to have access to the entire UDP packet, and to use data within the packet as its own parameters.

c. Artifice Base – General Functions

The artifice base provides several preset functions, but it also provides the user the ability to create his own functions that leverage the design of the artifice base. Like the artifice base itself, all functions loaded into the primary memory must adhere to a variety of restrictions, such as position independence and the preservation of particular registers. Position independence is crucial since there is no linking being performed by the system and no guarantee about the location of the primary memory. Sections of code may be linked together explicitly by the user, whether they are contiguous or in different

parts of memory. With regard to registers, the artifice base uses as a convention the EBX register to refer to its own data structures. All local variables are stored at relative offsets to the EBX register, which is loaded when the artifice base initializes. Users who wish to use the artifice base convention may do so; alternatively, they can use their own convention. If they do use the EBX register, they should also reset it to its original value once they finish using it to ensure that the artifice base may correctly reference its local variables.

The functions supported by the artifice base, both the built-in functions and the user defined functions, are available once the Artifice Base has been loaded. All supported functions require that the UDP header be formatted to the Start Packet specification. The fields that are common across all functions are shown in Figure 7:

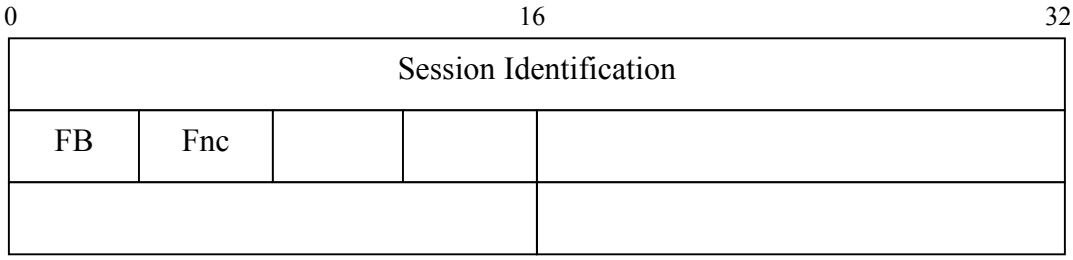

Figure 7. General Artifice Parameter Specification for Functions

The session identification field is a unique key that must match the session key currently defined in the artifice. This is used for all functions supported by the artifice base. The session key is stored as a variable within the artifice base, so the user is free to set up any type of coordination system he deems appropriate. For example, different users can be given different keys and instructed to change the session key when they want to transfer control of the artifice to another user. Or the user can establish two keys, one indicating that the artifice is currently being used, and another indicating that it is idle. The convention can be that an individual switches the session key to the in-use key prior to taking any action, and then switches it back once the action is completed.

The feedback (FB) field indicates whether the user wants feedback to be sent out to the network after a specific function has been completed. Because feedback

could be desired in conjunction with any function, the artifice base routine first executes the primary function, and then automatically executes the feedback function provided that the feedback flag has been set. Only the leftmost bit in the feedback field is used as the flag: a zero indicates that feedback should not be sent, a one indicates that feedback should be sent.

The function field (labeled as Fnc) is used to specify which function should be run. There are 16 possible trigger values, with 0, 1, and 2 pre-assigned to the standard functions of sending feedback, loading code, and setting new triggers. The remaining 13 values may be set to user-loaded code.

d. Artifice Base – Load Data Function

This function supports the loading of data into a specified location within the primary buffer. This function is available once the Artifice Base has been loaded, and it is reached by formatting the UDP header to the Start Packet specification. The fields that are needed in order to load data are specified in the header format diagram:

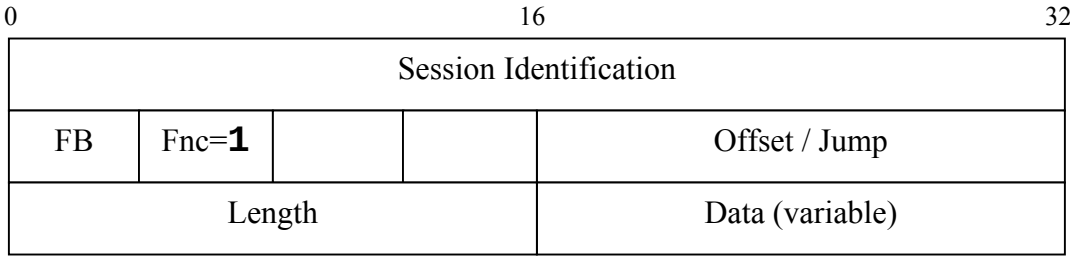

Figure 8. Artifice Parameter Specification for Load Data Function

The session identification field is a unique key that must match the session key currently defined in the artifice. The feedback (FB) field indicates whether the user wants feedback to be sent out to the network after the loading of data has been completed. Only the leftmost bit in the field is used: a zero indicates that feedback should not be sent, a one indicates that feedback should be sent. The load function is assigned the index of 1, so the function field (labeled as Fnc) must be `0x01`. The offset / jump field indicates where the data should be loaded. An offset of 0 indicates that the next byte following the artifice base code should be used for the first byte of data. The length field

indicates the number of bytes of data that should be copied. Neither negative offsets nor offsets that would result in code being loaded past the end of the primary buffer should be used; there is no error checking for these conditions. The data section of the packet starts at byte 10, and its size is specified by the length field.

e. Artifice Base – Set Trigger Function

This function supports the setting of a trigger for some portion of loaded code, thereby effectively creating a new function. This function is available once the Artifice Base has been loaded, and it is reached by formatting the UDP header to the Start Packet specification. The fields that are needed in order to load data are specified in the header format diagram:

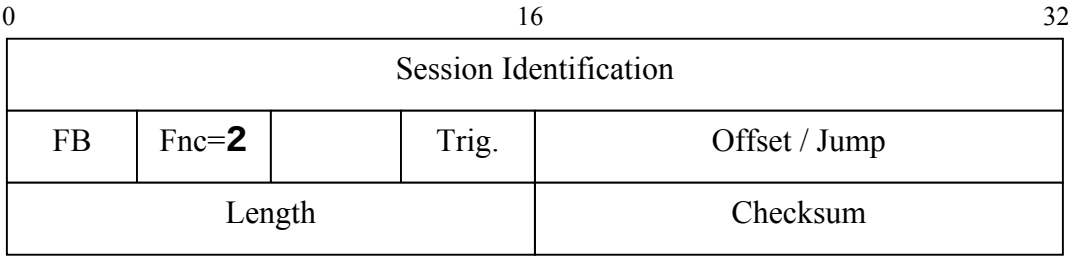

Figure 9. Artifice Parameter Specification for Set Trigger Function

The session identification field is a unique key that must match the session key currently defined in the artifice. The feedback (FB) field indicates whether the user wants feedback to be sent out to the network after the trigger has been set. Only the leftmost bit in the field is used: a zero indicates that feedback should not be sent, a one indicates that feedback should be sent. The set trigger function is assigned the index of 2, so the function field (labeled as Fnc) must be `0x02`. The offset / jump field indicates to what address the point of execution should jump when the trigger is specified in a future packet. For example, suppose that a trigger is set with a trigger of 8 and an associated offset is 22. When a future packet is sent that specifies 8 as the function number, then the point of execution will jump to offset 22. The length and checksum fields are used to ensure the integrity of the code associated with a trigger prior to the trigger being set, which can help to prevent a loss of control of the point of execution. The checksum fields

(length and checksum) are design placeholders, and are not actually used in the implemented demonstration.

f. Artifice Base – Feedback Function

This function supports the sending of feedback out to the network. This function is available once the Artifice Base has been loaded, and it is reached by formatting the UDP header to the Start Packet specification. The fields that are needed in order to load data are specified in the header format diagram:

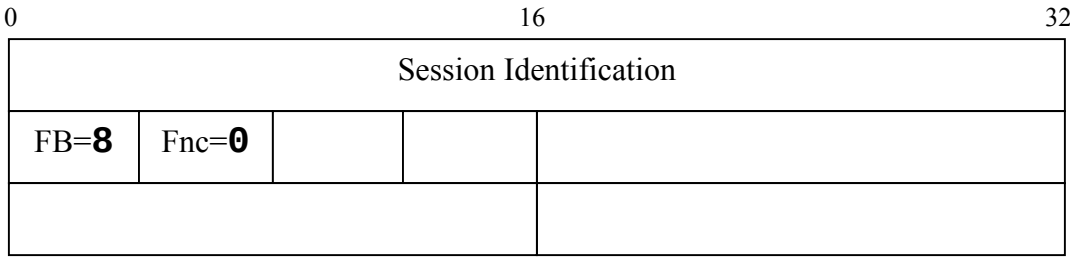

Figure 10. Artifice Parameter Specification for Feedback Function

The session identification field is a unique key that must match the session key currently defined in the artifice. The feedback (FB) field must be `0x08`, indicating that the user wants feedback to be sent out to the network. The `0x08` represents the setting of the leftmost bit to 1. The feedback function is actually a non-operational function; when no operation is combined with the optional feedback feature that is a part of any command, the result is the sending of feedback. Thus the function field (labeled as Fnc) must be `0x00` since the 0 function does nothing. The actual feedback that is sent depends upon variables that are located in memory within the primary buffer. These variables, which store the source and destination addresses as well as the packet payload, may be set by the user programmatically. Since the feedback function, when it is flagged, always executes immediately after the primary function being triggered, it is possible to program the primary function to store its results in the feedback buffer, which will be sent when the primary function completes.

THIS PAGE INTENTIONALLY LEFT BLANK

IV. ANALYSIS

A. WORK FACTOR

It would seem intuitive that the most potent types of attacks against information systems – the types of attacks that would be employed by professionals – would require the greatest amount of effort and skill, whereas the least potent types of attacks would require the least amount of effort and skill. In fact, such a straightforward correlation does not exist. For example, a cursory review of the vulnerabilities of a variety of systems and applications show that there are frequently exploits that are able to achieve only limited denials of service, yet are quite difficult to carry out. Conversely, writing a script that erases a hard drive on many UNIX systems (‘rm –rf /’) and giving it an attractive name could make a very potent but trivial to produce Trojan Horse. Another factor that makes correlation difficult is the fact that, with information systems, attacks can be automated. Thus, only one person needs to develop, program, and release an attack, after which executing the attack becomes significantly easier.

Indeed, in many respects, the most potent types of attacks are technically much easier to construct than less potent attacks. It is illustrative to compare the adaptable trap door-based subversion attack with a buffer-overflow penetration attack since there are so many similarities between them. While both need to ensure that the code they import is position independent, the buffer-overflow exploiter also needs to ensure that there are no bytes in the exploit code that are zero, since that would indicate the end of a buffer and therefore would not correctly trigger the buffer overflow. Also, since a buffer overflow typically overwrites an area that is in use (such as the stack), the exploit writer must also take care to ensure that her actions do not destabilize the system. Additionally, because of the many auditing mechanisms that could potentially be triggered by a penetration attack, implementing a penetration attack in a manner that avoids detection is much more difficult. In contrast, a trap door effectively operates as the operating system, making the triggering of auditable events avoidable. Another issue that makes penetration attacks more technically challenging is that a high level of privilege over the system is not a given: often an attacker must figure out how to escalate privileges in addition to gaining

initial access, which further complicates the attack as well as attempts to remain undetected.

Although subversion in many respects is technically easier than penetration, it does have some properties that make it more difficult to implement than a penetration attack. Primarily, subversion by definition requires a different attack vector than penetration. Penetration can occur at any time during the operation of a system, whereas subversion must use an attack vector that is related to the creation or maintenance of a system. Necessarily, a subverter must have a higher level of knowledge about how the system is developed, distributed, or maintained than an attacker using penetration. The attack vector for subversion is qualitatively different, requiring either some type of social engineering or meta-penetration, such as penetration of a development or deployment system, in order to affect the system it is supporting. While the skills required for subverting a system are different in quality, it would be presumptuous to conclude that such skills are more difficult to obtain or more difficult to effectively deploy. The fact that an organization attempting to conduct a subversion could use one party to design and create the artifice and another to insert it illustrates the potential ease with which an artifice could be inserted.

The construction of an adaptable subversion artifice is arguably technically more difficult than the design and construction of a fixed artifice. The primary reason for this is the addition of flexible code. The code that is used to bootstrap the system must be position-independent machine code, whereas a fixed artifice may be programmed with whatever high-level language is used to program the system. Also, while for a fixed artifice the artifice coder may take advantage of the knowledge of the linker to make reference to operating system variables and functions, any interfaces between the flexible exploit code and the operating system code must be discovered at runtime. Nevertheless, in the author's judgment, the estimate posited by Myers that subversion is within the skill level of the average undergraduate computer science major is accurate (though a college education is certainly not necessary). Although the creation of this thesis and the associated demonstration lasted 9 months, the actual implementation of the artifice took only a couple of weeks. The most significant technical challenge of the project was

deciding what tools to use and becoming familiar with the tools and development environment. In this regard the effort was similar to many programming projects, where an unfamiliar development environment is generally a significant initial hurdle.

B. IMPACT ON THREAT MODEL

In 2000, the Office of the Assistant Secretary of Defense (Command, Control, Communications and Intelligence) OASD (C3I) published a report [OASD00] that outlined a set of recommendations intended to mitigate the insider threat to DoD information systems. The report defines "insider" and sets up a framework for understanding and analyzing the insider threat, and it divides its recommendations into short-term and long-term actions that should be taken to mitigate the risks. It is said that when producing software, it is very important to remove "bugs" early in the development process, preferably in the design phase, since it is many times cheaper to remove a bug at that point than once the bug has been written into the code. Risk mitigation has a similar characteristic in that a small misunderstanding of the extent of the threat can result in a tremendous amount of extra effort without a corresponding reduction of risk.

In the framework section of the report, the threats from insiders are characterized. In particular, the report specifically addresses the potential threat from the vendors who supply COTS products to the DoD. While the report notes that little is known about the individual developers of such products and therefore no measure of trustworthiness can be attributed to them, the report downplays the feasibility of an individual developer acting on his potentially malicious intentions:

> …individual developers of COTS products who have malicious intentions would have an extraordinarily difficult task to target a particular customer because COTS products tend to be produced in large quantities and shipped to customers as an activity that is independent of the individual developer. The developer with malicious intentions would have to deliver the same product to all customers while retaining the ability to isolate a particular customer for exploitation.

The report does grant that COTS systems do contain errors and that it is extremely easy to demonstrate the ways in which such systems are insecure. Nevertheless, as explained

previously, there is a fundamental difference between a vulnerability caused by a bug and a vulnerability intentionally inserted as a trap door.

The task that the report dismisses as too difficult is precisely the task that this thesis proves is quite easy. The task is to create an adaptable trap door that can be safely inserted into every distributed copy of a system, since there is a vanishingly small chance that the artifice will be detected. This trap door can be selectively activated just for the particular target customer, and although the trap door is in every system, the exact capabilities desired for any particular target may be inserted into the particular target customer's system at the time of activation.

The report does not explain in detail exactly how the attributes and assumptions spelled out in the framework led to the specific actions that were recommended. The flavor of the recommendations is, however, clear. Many of the recommendations focus on measures that overlook the underlying insecurity of systems, and instead focus on ways to append security. These additive tactics include using defense in depth, adding additional security tools, introducing additional layers of cryptography, increasing the number of attributes that are audited and measured, and attempting to detect and react to anomaly and misuse detection. One action item labeled "System security architecture" listed several important features of a secure system that warrant additional research, such as authentication, access control, system integrity, and a bi-directional trusted path, yet left out the concept of providing assurance for the system itself. Although such a property may be implied, the bullet stating that the system should "continuously check system integrity, to prevent violations of integrity caused by … malevolent software…" seems to indicate that a high-assurance system is not being envisioned.

The report does make one recommendation that pertains directly to potentially subverted systems. It identifies that the acquisition process needs to be re-implemented to ensure that program managers evaluate their potential vulnerability to subversion. The ability to mitigate this threat, however, is misguided. For example, the report states "If the vulnerability would jeopardize a critical Defense capability, additional security would be instituted." Of course, adding an additional security measure is ineffective against a subversion. Another statement from the report explains that "Action … is needed to

provide … an identification of otherwise obscured system vulnerabilities." Given the known difficulty of detecting a subversion artifice, it is unclear how such vulnerabilities will be identified.

The report warns that the insider risk is real and significant, yet it overlooks the threat with the biggest potential and most catastrophic consequences. More alarmingly, it provides a large array of recommendations, with a typical recommended action being very costly, focused on detection, and completely impotent against subversion. Although prevention is traditionally more efficient than either detection and response, and it is incidentally the only way to guard against subversion, the report provides few recommendations with a preventative focus.

THIS PAGE INTENTIONALLY LEFT BLANK

V. COUNTERING THE SUBVERSION THREAT

A. THE PROBLEM OF IDENTIFYING AN ARTIFICE

Determining whether software contains a trap door, although it sounds like a straightforward task, has been shown by history to be an exceedingly difficult task. Perhaps the most illustrative analogy is that finding a backdoor in software is equivalent to determining whether software contains a bug. In fact, it is most frequently software bugs that lead to system vulnerabilities, and a trap door could in fact be created by intentionally inserting a software bug. Anyone who is familiar with software and the propensity for software to contain bugs will realize that this is not a trivial problem to solve.

One of the factors that make it difficult to find a trap door is simply that there is so much code to examine. The Microsoft NT operating system is estimated to have approximately 50 million lines of code. Additionally, in order to determine that there is no trap door it is not sufficient to find just one bug, it is necessary to find (and remove) every bug. Only one flaw need exist to create a trap door. Unfortunately, determining whether malicious code exists is more difficult than finding all the needles in the proverbial haystack, for often it is difficult to know what a needle looks like.

One of the arguments made by the open-source community is that even with a great deal of code, it is possible to find most of the bugs in code since there are so many people examining the code. This idea is most famously expressed by Eric Raymond in his essay "The Cathedral and the Bazaar" [RAYM99] by the phrase, "Given enough eyeballs, all bugs are shallow." He asserts that this property of software developed with the "bazaar" model of development makes this type of software less buggy, and also reduces the amount of time any bug would likely remain in the system. Open source advocates also claim that 'Trojan horse' code will not survive in open-source software due to these many eyeballs, often citing an incident in 1999 when the TCP-Wrappers source code was infected with a Trojan horse and quickly discovered by the open source

community. However, John Viega, co-author of "Building Secure Software", points out that this was a glaringly obvious example of a Trojan horse [VIEG01].

Viega proposes that the often cited 'many eyeballs' phenomenon is a myth since it commonly doesn't occur. In his estimation, just because many people may view some source code doesn't mean that the security vulnerabilities will be uncovered. The reasons for this are many, including: most often the only eyeballs are from those who want to modify code (which may be very few to none), many coders don't understand security issues and may not care about them, and many security issues are very subtle and are not readily apparent even for a motivated and educated observer. Viega cites a real-world example of how difficult it is to detect an exploitable flaw in software, even when the amount of code being examined is very small. The source code for wu-ftp, a very common and popular piece of software, has been open and publicly available for over a decade and has been examined extensively by security experts for security vulnerabilities. In fact, at one point a security tool had flagged part of the wu-ftd code as potentially vulnerable to a buffer overflow attack. Security experts who reviewed this particular portion of the code determined that the potential flaw could not be exploited. Despite this fact, more than a year after this code was reviewed, a buffer overflow exploit for this same code was published. This example shows that detecting a security vulnerability can be extremely difficult even when only a small amount of code needs to be analyzed. This property, when combined with the considerable amounts of code that make up even relatively simple systems, illustrates that finding a trap door inserted in the source code of a large and complex system is virtually impossible.

The fact that the trap door need not even be in the source code at all makes the virtually impossible task of finding the door even more difficult. The trap door can be pre-compiled, provided there is a means of getting the compiled binary with the trap door code into the target system. Karger and Schell hypothesized an even more insidious scheme [KARG02]: store the directions for installing a trap door not in the source code, but in the compiler program that translates the source code into an executable file. In this case, there is no source code to review; one must ensure that the compiler program does not have the capability of installing a trap door. The Karger and Schell publication

inspired Ken Thompson to install such a self-compiling trap door into an early version of UNIX. The existence of this trap door was first revealed in 1984 during the speech Ken Thompson gave upon receiving the Turing Award [THOM84]. Ken Thompson spoke directly to the trust one can place in software:

> The moral is obvious. You can't trust code that you did not totally create yourself. (Especially code from companies that employ people like me.) No amount of source-level verification or scrutiny will protect you from using untrusted code.

While inspection of source code alone will not provide protection from untrusted code, a methodology that provides a basis for trust in information systems does exist. This methodology, currently outlined in the Common Criteria [CC99], encompasses a number of principles, which, when applied in combination, provide evidence that a system provides a certain level of assurance. While source code verification alone cannot provide a reasonable level of certainty, a combination of measures may. For instance, the following list of measures provide a considerably higher level of assurance than source-code verification alone:

- Conducting background checks on programmers
- Using only a custom-created compiler
- Using a distribution method that ensures that the delivered product is the same as the developed product
- Minimizing the security components of the system
- Ensuring that the security components map precisely to a formally specified information flow policy
- Organizing the security components in such a way that their functionality may be fully understood

This last measure will be explored in detail in the next section.

B. THE ROLE OF MODULARITY, INFORMATION HIDING, AND LAYERING

In dealing with the problem of subversion, the challenge is to be able to ascertain with certainty that no artifice has been inserted into the system. The only way to be certain that no artifice has been inserted into the system is to have a complete understanding of the system. Every line of code in the system and every possible interaction between different parts of the system code must be completely understood, and each line of code and interaction must ultimately support the specification around which the system is designed. If there are any deficiencies in this understanding, there is the possibility for the insertion of malicious code.

Modularity, Information Hiding, and Layering are the tools used to gain a complete comprehension of large system. When faced with the possibility of subversion, the solution ultimately always boils down to the need for a complete understanding of the code. When the code is small enough, an understanding of the overall intent and mechanics of such code is possible. When the size of the code grows, however, the natural complexity of the system quickly makes it impossible for any one person to be able to understand the code in its entirety. The solution to this issue is to divide the code into separate parts, and to ensure that the parts are small enough that each part may be understood in its entirety. The issue at that point is that, since the parts interact with one another, one must have the ability to understand the interaction between parts. These parts are called modules, the interactions between the parts is known as layering, and a critical aspect of modules is that the internal state and contents of the modules must be hidden from other modules. Only by limiting the interactions between modules can we hope to have an understanding of the overall system behavior.

1. Common Criteria Guidance

The concepts of modularity, information hiding, and layering are at the foundation of good software engineering practices, so it is hardly surprising that the Common Criteria, the generally accepted guidelines for building high-assurance computer systems, requires the incorporation of the principles of modularity and layering [CC99].

For example, the Common Criteria requires the use of modularity and layering in the internal structure of a system's security functions at the highest levels of assurance (EAL5 through EAL7). For a system to meet EAL5 requirements, the security functions must be designed in a modular fashion in order to avoid "unnecessary interactions between the modules." For a system to meet EAL6 requirements, it must (in addition to the modularity requirement) structure the security portion of the system in a layered fashion, minimizing the mutual interaction between the layers in the design and reducing circular dependencies. It must also reduce the complexity of the functions that are directly responsible for access control and/or information flow. In order for a system to meet the highest level of assurance requirements, EAL7, it must reduce the complexity of the access control and/or information flow functions (potentially by removing functions that are not critical to the enforcement of such functions) to the point that they can be fully analyzed.

The Common Criteria does point out that the overall goal of these measures is to reduce the complexity of the functions being analyzed, which leads to an increase in comprehensibility and ultimately a higher level of assurance that the functions accurately and completely fulfill their requirements. However, it provides only a high level description of how these organizational principles can assist with this goal. This chapter will explain why these principles are critical to the construction of a high-assurance system, how they can assist in assuring that no artifice is inserted, and the desirable benefits a system may gain through the use of these principles, regardless of whether it is designed for high assurance.

It is important to note that although the principles of modularity and layering are necessary for the construction of systems that provide the highest levels of assurance, they are by no means sufficient. Within the Common Criteria, for example, these principles apply only to a single family of requirements within the class of requirements for how a system is developed (the particular family is called "Target of Evaluation Security Functions (TSF) Internals"). Also within the class of development requirements are families of requirements that pertain to functional specification, high-level design, low-level design, security policy modeling, etc. In addition to requirements that pertain to

system development, the Common Criteria has additional classes of requirements for such areas as configuration management, delivery and operation, lifecycle support, and testing. Assurance cannot be provided in a piecemeal fashion: only when multiple attributes are considered in parallel can confidence be achieved. For example, even if the development of a system were conducted perfectly, the ability of a malicious entity to modify the system during delivery makes the assurance measures taken during development meaningless. This characteristic of assurance is reflected in the Common Criteria by the fact that an increase in overall assurance level can only be achieved by a combined increase in measures taken across requirement classes. Although modularity and layering can help to ensure that a system is not subverted during development, meaningful assurance is possible only through the application of a wide variety of measures throughout every stage in a system's lifecycle.

2. Modularity and Information Hiding

Parnas produced the seminal works on the topics of modularity and information hiding thirty years ago [PARN72]. The original motivation behind the idea of modularity is related more to the efficient construction of software and the ability to redesign the software for other uses than for security. These goals, however, overlap with the goals implicit in creating high-assurance software. The overall goal is the same: rendering the system as a whole understandable.

Parnas's original description of a module is that of a work assignment: a task that could be given to an individual programmer who could grasp the entire scope of the assignment, produce the code for the module in question, and deliver the code for the module so that it could be recombined with other modules. This effort to efficiently divide the labor of those working on a large project ends up having a host of other benefits.

In order to divide the work up in an efficient manner, the problem must be viewed in terms of what is likely to change frequently versus what is likely to remain constant. The aspects of a module that remain constant, Parnas concluded, should make up the interfaces between different modules. The aspects of the module that change frequently should be represented by the internal databases managed by the module. Thus, a

programmer assigned to a module encodes the entire module to comply with a static interface knowing that the interface will be able to be used by modules written by other programmers. This way of designing a system has the effect of hiding the information internal to the module, which has additional advantages. Information hiding, for example, allows for the internal structure of a module to be revised without changing the overall capabilities of the system, and it also allows the module to be tested independently from any other modules. Arranging the modules into a hierarchy enables one to understand the relationship between modules, thereby ensuring that work is not duplicated between modules. It also allows a new programmer to understand the overall structure of the system. Thus, correctly decomposed, a system will have small modules, each of which is fully comprehensible by the person who programmed it. Also, the number of interactions between modules is significantly limited by the use of information hiding, enabling one to understand the overall behavior of the system more easily.

These ideas have been so successful, in fact, that an entire generation of programming languages has been developed to support these ideas. The object-oriented approach is today seen as the standard way to develop software. Just because the tools are available, however, does not mean that the concepts will be adhered to.

2. Layering

Once a system has been decomposed into small enough pieces, the pieces—or modules—must then be combined together to form the entire system. Without interactivity among modules, the usefulness of the system would be indeed limited. Without a structure that defines the potential communication paths between modules, understanding the overall system would quickly become unmanageable.

Dijkstra originally observed the utility of adhering to a strict program of structuring in the design of the THE system [DIJK68]. He observed that by breaking the system into small enough layers, and testing each individual layer with sufficient rigor to prove to oneself that the individual layer was correct, that it was possible via the transitive nature of the layers to prove the correct operation of the entire system. The THE system was composed of five layers. Each higher layer depended upon each of the lower layers. Dijkstra commented that the use of this layering was imperative to

constructing a system that could be totally understood. While many have commented that the system Dikstra built was relatively simple, and therefore may not apply to larger efforts, Dijkstra would have countered that a larger project would have even more of a need for correct structuring.

One of the important characteristics of a layered structure is its ordering. While Dijkstra organized the THE system into a total ordering, other efforts have successfully used a partial ordering of modules. A layered structure implies that the layers upon which any given layer rests are independent of the higher layers: each layer is able to exist independently since it is the foundation for the next higher layer. The lack of upward dependencies reflected in this design is very difficult to achieve. In fact, even when efforts are made to limit the interactions among modules by organizing them into layers, these rules are regularly circumvented when the implementation is actually carried out. Perhaps the best example of circularity in the design of an operating system is illustrated by the common interaction between the virtual memory management system and the file system. In order for the virtual memory manager to swap the contents of memory to disk, it relies on the file system to provide the storage for these pages. In a symmetric fashion, in order for the file system to manipulate large files (larger than physical memory), it relies on the virtual memory manager to transparently handle the large files. Thus, a cyclical dependency between the two systems exists.

It may not initially be apparent why this situation is problematic. One might even argue, for example, that such a design has the advantage of being more compact since code is being reused. However, the increased complexity that is created due to this design has a variety of characteristics that make it undesirable when understandability is a goal. One of the issues with cycles has to do with error states. With a circular dependency, an error state in one part of the system can propagate through the cycle and create an endless loop. An example of this issue appeared in the design of the auditing system for VAX VMM security kernel [SEID90].

Another issue with circular dependencies is that the modules that take part in creating the cycle can never be evaluated or tested independently. In effect, circularity spoils the advantages gained by the use of small information-hiding modules: to be able

to understand them completely, to be able to test them independently, to be able to modify the internal representation of modules without impacting the correctness of other modules. The cycle of dependencies creates a condition whereby all of the modules that are involved in the cycle effectively create one large module, which can therefore not be completely understood. One of the important properties of a well-structured system is that a subset of the system can be considered independently of the entire system, yet still be considered complete.

Perhaps most importantly, the existence of cycles prohibits the applying of formal (i.e. mathematically-based) methods to assist with the understanding of the system. A system that contains cycles may never formally correspond to a higher-level mathematic description of a system due to the critical requirement for transitivity. Any logical system that mapped onto an implementation that contained cycles would be equal to the logically flawed application of circular reasoning. These principles inherent in a hierarchical design that preserve logical transitivity and the ability to have system subsets apply directly to the concept of a security kernel.

While having a small security kernel is important, especially due to the costs associated with verification, there is still the need for additional security services outside of the kernel to create a usable system (such as the authorization system, a trusted path, etc.). These additional components in addition to the security kernel are known as the "trusted computing base" or TCB. Using a hierarchically-structured design allows these other components to interface with security kernel in a logically sound manner. Although these components may not correspond to the formal specifications, they may still be designed using the same principles and techniques. Without a hierarchical design, the interactions between these modules and the security kernel could be complex and likely poorly understood.

3. Applying the Principles

Applying the concepts of modularity, information hiding, and layering is not always easy or obvious, especially when the areas of interest are inherently complex, such as the kernel of an operating system. Many consider such efforts to be wasted, since the design decisions are difficult and perhaps not intuitive [LAMP83]. Nevertheless,

many techniques have been developed that permit the application of these principles, and they have been successful conceptually in several projects despite their lack of adoption in the marketplace.

The fundamental intellectual work done on applying these principles to operating system design was performed by Janson in his doctoral thesis [JANS76]. His ideas were later adopted in the redesign of the Multics kernel, in the creation of a Virtual-Machine Monitor security kernel for the VAX platform, and in the GEMSOS kernel. Janson's thesis describes the construction of a virtual memory mechanism using a strict partial ordering among the modules, and the concept of type extension to enforce the information hiding aspects of the modules.

The Janson thesis describes the connections between modules by characterizing the different types of dependencies that may exist. A dependency is considered to exist between two modules, say A and B, only if the correct behavior of A depends upon the behavior of B. To create a partial ordering, all upward dependencies must be eliminated. There are a limited number of inter-modular dependencies that may occur between modules:

1. Message Only – Module A sends a message to module B, does not abandon control, and does not expect a response.
2. Quiescent Signaling – Module A sends a message to module B, abandons control, and does not expect a response (and therefore is in a quiescent state).
3. Non-Quiescent Signaling – Module A sends a message to module B, abandons control, but does expect a response (and therefore is in a non-quiescent state).
4. Quiescent Transfer of Control – Module A transfers control to module B, and does not expect a response (and therefore is in a quiescent state).
5. Non-Quiescent Transfer of Control – Module A transfers control to module B, but does expect a response (and therefore is in a non-quiescent state).
6. Call – Module A explicitly calls module B and awaits a response (and therefore is in a non-quiescent state).

As defined by Janson, the interactions described by points 3, 5, and 6 are "invocations," whereas points 1,2, and 4 are "notifications." Invoking another module implies a dependency, since a response is always expected. A compiler can automatically identify dependencies created by upward calls. Unfortunately, a compiler can not necessarily prevent upward invocations since differentiating between 2 and 3 or between 4 and 5 requires understanding the intent of the module, i.e. knowing whether the module remains in a quiescent state or expects a response.

Janson describes the second form of dependency as originating from the information that is passed between modules when an interaction takes place. Parameters that are passed can either be values, or they can be references. In the case of passing a value, a module may be dependant upon another module if it trusts that the values passed by that module are a particular type or range without checking the value. Object-orientation is generally unhelpful in identifying value parameters. While languages may be able to provide some help with checking for the type of values, they do not assist with the checking of ranges. Object-orientation is generally more helpful for the control of references. Since the compiler understands the hierarchy of the modules, it can ensure that a reference to a "higher-level" module can never be acted upon. While the higher-level objects may be referred to, this is done only in terms of the reference as a value: the object itself cannot be accessed.

Invocations and notifications are categorized as component dependencies. In total, there are five types of dependencies described:

1. Component Dependencies—a module depends upon its own components, which are other modules.
2. Map Dependencies—a module must keep track of the objects that are its components, but the maps that allow the component objects to be tracked must be kept in some kind of information store. Thus the module is dependant upon the objects providing storage for the mappings.

3. Program Storage Dependencies—every object that may be called has some kind of code associated with it, and that code must be stored in some object in order for it to be executed.

4. Address Space Dependencies—the objects that make up the execution environment of a module need address space in order to be executable. Thus it is dependent upon the information container that provides this address space.

5. Interpreter Dependencies—any module has code that needs to be executed by a processor, and is thus dependant upon the module that provides the notion of a processor. Since every object does not have an individual physical processor, it is dependant upon the module that provides the virtual processor.

Based upon the research done in the Multics redesign project [SCHR77], it was concluded that the circular dependencies created due to the mapping, program storage, and address space are easy to break once their contingencies are recognized. The primary principle used to solve these issues is the use of core segments: areas of memory that use a fixed amount of space and are permanently stored in main memory. The most complex problems arise from the handling of exceptions, since they often arise when low-level modules are trying to communicate with high-level modules. Often the instinct of a designer in this case is to have the lower-level module call the higher-level module. This dependency loop may be removed with the use of additional hardware interrupts or with the careful use of software signaling.

A common strategy that is used to break dependency loops caused by interpreter dependencies is the use of modules on two-levels that work together to provide the illusion of infinite resources when finite real resources are available. Typically the lower-level module provides a fixed number of resources, and is closely connected with the underlying hardware. The higher-level module then provides a similar resource, but it gives the illusion of unlimited resources by multiplexing the lower-level resources. This strategy was used in the design of both a virtual processor system (by Reed [REED76]) and of a virtual memory system (by Janson).

One of the issues encountered in the implementation of a two-level virtual processor is that there are times when the low-level virtual processor must change the state of the upper-level process. However, the low-level virtual processor is not allowed to know anything about the higher-level processors. What is needed in this case is a mechanism that allows upward communication without the sending party having any knowledge about the receiving party. Reed's implementation enabled this by using a synchronization protocol based on eventcounts [REED79]. This allows a processor interrupt to be tied to an eventcount, so that a higher-level processor can be interrupted when the eventcount reached a certain value. This allows for upward transfer of control that would otherwise result in an upward dependency.

4. Addressing Performance Concerns

It is a generally accepted assumption that any increase in security must be accompanied by a decrease in performance. In the same vein, the performance of a highly structured system is often assumed to be inadequate since performance-improving shortcuts are generally forbidden. Undoubtedly there is some basis underlying this assumption of unacceptably poor performance. Even if the vast majority of previous efforts to build highly structured systems have resulted in poor performance, this does not imply that building such a system is impossible. There are probably a variety of causes that have lead to the negative perception. One possible factor may have been that many such efforts have been research projects, which can be considered successful without displaying practical performance characteristics.

In reality, there is evidence suggesting that it is possible to create highly structured systems that also provide adequate performance characteristics. Adequate performance is, of course, relative, and implies that the systems in question did not incur an intolerably high overhead. In fact, the systems that serve as positive performance examples were developed and considered usable on now antiquated hardware. Given the state of continuing improvements in hardware performance, the minor penalty that may be incurred by creating highly-structured code is more than made up for by the advantage of comprehensibility.

There are a number of design factors that impact performance for any system; special consideration of these factors should be taken in the case of designing a highly-structured system however, since the workarounds and shortcuts that may be available in less structured systems will not be possible.

The choice of language is an important consideration since it directly impacts the efficiency of the code, but the quest to build comprehensible, bug-free code often shifts the focus toward languages that support features such as strong data typing. Unfortunately, there is often no perfect choice: the most popular languages typically provide the most efficient compilers but lack the features of strong data typing. Conversely, languages that provide strong data typing features tend to be less popular and therefore less effectively optimized.

Implementing components of a system in hardware is a proven way of achieving performance gains. A system design that is able to take advantage of hardware support will realize a considerable performance advantage over a design that implements all of its components in software. Specific types of hardware support can be especially helpful with regard to supporting security policies, including process management and switching, memory segmentation, input/output mediation, and execution domains.

It should be noted that attempting to make comparisons of performance is notoriously difficult. Comparisons between modern-day operating systems, for example, usually indicate more about who is doing the testing than about the relative objective performance of the measured systems. Also, it is readily apparent, as Parnas noted, that software may be written to perform poorly regardless of the software engineering techniques used. So the raw performance measurements are not very telling, but the important characteristic to ascertain is whether or not it is even possible to implement a system based upon these principles that performs adequately. Although the number of systems that have been created using these principles is small, the evidence in sum is positive.

a. Multics

The first system considered is Multics, specifically the version of Multics that was the product of a redesign effort to improve security. After the redesign of the Multics kernel to improve its security, the designers concluded that the use of simpler and more modular designs did not result in any significant loss of performance. In one of the areas where performance did decrease, the reason for the decrease was clear since the code was rewritten in a high level language after initially being programmed in assembly language. This type of performance impact is consistent across a wide range of examples, and is not necessarily indicative of a deficiency in the structuring techniques.

b. VAX VMM Security Kernel

A second project, the VAX Virtual-Machine Monitor security kernel [KARG90], provided sufficient performance for production-quality use. In practice, it was able to support its' own development team, and was able to support a large number of time-sharing users running multiple operating systems on a single CPU. The designers noted that although achieving a high level of performance was difficult, they were able to analyze their system sufficiently well to optimize the portions of the system that were most performance-critical. The issue in this case was less about the overall structure of the system, and more about the language used for coding and ensuring that direct interface to the hardware was optimized. One might suppose that the application of these engineering principles to creation of a Virtual-Machine Monitor would be easier than to the creation of a general-purpose operating system. Although this is true, the performance success of the virtual-machine monitor is in a way an even stronger validation of the concepts, since any deterioration in performance would be exaggerated in the full-fledged operating systems running on top of the virtual machine. The performance success of this project is all the more impressive given that the machine successfully limited the bandwidth of covert channels using fuzzy time [HU91], which did have a performance impact unrelated to the structural design.

c. GEMSOS Kernel

The third system for which performance has been evaluated is the GEMSOS kernel, which was a commercial operating system that was successfully

evaluated against the Class A1 TCSEC requirements and fully utilizes the structuring techniques being considered. An early version of the GEMSOS kernel was tested both for throughput characteristics – especially with regard to the improvements gained via the use of multiple processors – and for the ability to support real-time processing. Both of these tests showed that the GEMSOS kernel, even in an early non-optimized version, had adequate performance [SCHE85]. This performance was achieved despite the use of Pascal as the programming language, which did not provide the most efficient compiler available at the time. One noteworthy aspect of the design was that the use of multiple processors in the system achieved nearly linear increases in performance.

d. L4 Microkernel

The fourth and last system considered is a micro-kernel structured system. Although the primary focus of the system is not security, there is a strong synergy between the previous efforts, which focus on using rigorous structuring techniques to achieve a higher level of assurance regarding security, and the micro-kernel philosophy of operating system design. Both schools of thought originated around the same time (though independently), and both understood the value of (and were designed to support) a highly-modular design. Also, many of the early attempts to produce working micro-kernel architectures resulted in systems with unacceptably poor performance.

Efforts to improve the design of the micro-kernel have continued, however, and the current "second generation" of microkernel implementations has resulted in significantly better performance. Of note is the L4 micro-kernel, which was tested extensively by Härtig et al [HART97]. Modifications were made that allowed the Linux operating system to run on top of the L4 micro-kernel, and the performance of this system was then compared with the performance of native (monolithic) Linux using a variety of benchmarking techniques. The L4-based Linux platform, named L^4Linux, was also compared with the performance of a first-generation micro-kernel, specifically MkLinux, which is a version of Linux that runs on top of a micro-kernel based on Mach. These tests showed that even the non-optimized L^4Linux platform suffered only a 5-10% performance penalty from native Linux performance, whereas MkLinux registered a much more significant penalty of between 25-50%. The micro-kernel approach of the

second-generation kernels the first step toward a system that fully reflects the ideal of modular design. Its performance results are then merely suggestive of what may be possible in a completely structured system.

These four examples show that a highly-structured system design for an operating system does not necessarily imply that the system will perform poorly: designs that are highly-structured, highly-assured, and high-performance do exist, and can be built.

THIS PAGE INTENTIONALLY LEFT BLANK

VI. CONCLUSIONS

As the Information Age takes hold and the world becomes increasingly connected, the dangers that arise from those interconnections multiply. The U.S. Department of Defense is pursuing a strategy of cyber network warfare, hoping to leverage computer networks to better enable military capabilities. Yet each new additional capability represents an additional information asset that needs to be defended as well as a potential new avenue of attack. In the corporate world, the desire for more productivity that is being driven by global competition is encouraging an increase in the use of computer networks to leverage employee productivity and to improve services. This might lead a power company, for example, to consolidate power control centers and to connect power control center databases with corporate networks to improve marketing efforts. These consolidated control centers, of course, create a much more lucrative target, and connecting the control center to the corporate network creates a new avenue for attack.

Increasingly, the mitigation strategy against these risks has been to add more "solutions" to the mix. Now instead of a just a server, there are firewalls, intrusion detection systems, anti-virus products, public-key encryption systems, etc. Often such solutions consolidate security management responsibilities, creating a situation where the new systems need to be more trustworthy than the systems for which they provide protection. And each new system brings its own management challenges and maintenance requirements. Ironically, most of these systems are based on a commercial off-the-shelf operating system foundation, just the system whose deficiencies were part of the reason for adding additional security measures in the first place. This mitigation strategy has even become policy for the Department of Defense [DOD02].

Unfortunately, this strategy does nothing to mitigate against the threat of subversion, the type of attack that has the greatest potential for damaging the nation's critical infrastructures and compromising national security. Subversion, which is no more difficult to implement than other types of attacks, is unaffected by additive security measures.

In March of 2000, Steve Cross, director of the Software Engineering Institute at Carnegie Mellon University (home of the CERT Coordination Center), provided testimony before the Senate Armed Services Committee: Subcommittee on Emerging Threats and Capabilities [CROS00]. After providing an overview of the vulnerabilities of the Internet, and the impact of known security breaches, he recommended solutions. One of his recommendations concerning research and development was to: "leverage past investment that has produced an extensive, but little used, body of knowledge in rigorous methods for system analysis and design…."

The techniques for developing systems that provide a high level of assurance of their correct operation already exist. Without application of these techniques toward the creation of commercially-available products, however, the never-ending battle of wits between attackers and defenders will remain the only game in cyberspace. The use of such products is inevitable, for the day will come when the cost of the "battle-of-wits" will simply be unsustainable. The only question is when such efforts will be reinvigorated. Hopefully, it won't take some incident of cyber-terrorism to provide the awareness and resolve necessary to counter subversion.

There is at least one attempt being made to capture the lessons learned from past high-assurance development efforts and create a worked example that can be used to educate future students and developers. The Center for INFOSEC Studies and Research (CISR) is embarking on a project to create a "Trusted Computing Exemplar" that will serve as such an example [IRVI03].

LIST OF REFERENCES

[ANDE72] Anderson, J.P., Computer Security Technology Planning Study, Technical Report ESD-TR-73-51, Vol. II, Air Force Electronic Systems Division, October 1972. (NTIS document number AD758206).

[ANDE02] Anderson, Emory A., *A Demonstration of the Subversion Threat: Facing a Critical Responsibility in the Defense of Cyberspace*, Master's Thesis, Naval Postgraduate School, Monterey, CA, March 2002.

[BELL73] Bell, D.E. and LaPadula, L.J., Secure Computer Systems: Mathematical Foundations and Model, Technical Report M74-244, MITRE Corporation, Bedford MA, 1973.

[BIBA77] Biba, K.J., Integrity Considerations for Secure Computer Systems, MTR3153, MITRE Corporation, Bedford, MA, April 1977.

[BRIN95] Brinkley, D. L., and Schell, R. R., "What Is There to Worry About? An Introduction to the Computer Security Problem," in *Information Security: An Integrated Collection of Essays*, ed. Abrams, M. Jajodia, and Podell, IEEE Computer Society Press, Los Alamitos, CA, pp. 11-39, 1995.

[CC99] ISO/IEC 15408 – Common Criteria for Information Technology Security Evaluations, version 2.1, August 1999.

[IRVI03] Irvine, C.E., Levin, T.E., and Dinolt, G.W., "Trusted Computing Exemplar Project," White Paper, The Center for INFOSEC Studies and Research, Monterey, CA, http://cisr.nps.navy.mil/projecttrustcomp.html, September 2002.

[CROS00] Cross, Steve, Testimony before the Senate Armed Services Committee: Subcommittee on Emerging Threats and Capabilities, http://www.cert.org/congressional_testimony/Cross_testimony_Mar2000.html, 1 March 2000,.

[DIJK68] Dijkstra, E. W., "The Structure of the THE Multiprogramming System," *Communications of the ACM*, Vol. 11, No. 5, pp. 341-346, May 1968.

[DOD02] Information Assurance, DOD 8500.1, U.S. Department of Defense, October 2002.

[DOD85] Trusted Computer System Evaluation Criteria, DOD 5200.28-STD, U.S. Department of Defense, December 1985.

[DOD88] Security Requirements for Automated Information Systems (AISs), DOD 5200.28, U.S. Department of Defense, March 1988.

[FRAI83] Fraim, L.J., "SCOMP: A Solution to the Multilevel Security Problem," *IEEE Computer*, July 1983.

[HART97] Härtig, H., Hohmuth, M., Liedtke, J., Schönberg, S., and Wolter, J., "The Performance of μ-Kernel-Based Systems," *16th ACM Symposium on Operating System Principles* (SOSP), St. Malo, France, October 1997.

[HU91] Hu, W.M., "Reducing Timing Channels with Fuzzy Time," *Proceedings of the IEEE Symposium on Research in Security and Privacy*, Oakland, CA, pp. 8-20, May 1991.

[JANS76] Janson, P.A., *Using Type Extension to Organize Virtual Memory Mechanisms*, PhD thesis, Department of Electrical Engineering and Computer Science, MIT, Cambridge, MA, 1976. (Published as Technical Report MIT/LCS/TR-167, Laboratory for Computer Science, MIT, Cambridge, MA, September 1976).

[KARG02] Karger, Paul A. and Schell, Roger R., "Thirty years later: Lessons from the Multics security evaluation." *ACSAC: 18th Annual Computer Security Applications Conference*, Las Vegas, NV, USA, October 2002. (Includes by the same authors, Multics Security Evaluation: Vulnerability Analysis, ESD-TR-74-193, Air Force Electronic Systems Division, June 1974).

[KARG90] Karger, Paul A., Zurko, Mary Ellen, Bonin, Douglas W., Mason, Andrew H., Kahn, Clifford E., "A VMM Security Kernel for the VAX Architecture," *Proceedings IEEE Symposium on Research in Security and Privacy*, Oakland, CA, pp. 2-19, May 1990.

[LAMP83] Lampson, B.W., "Hints for computer system design," *ACM Operating Systems*, Rev.15,5, pp 33-48, October 1983. (Reprinted in *IEEE Software* 1,1, pp 11-28, January 1984).

[LOSO98] Loscocco, Peter A., Smalley, Stephen D., Muckelbauer, Patrick A., Taylor, Ruth C., Turner, S. Jeff, and Farrell, John F., "The Inevitability of Failure: The Flawed Assumption of Security in Modern Computing Environments," *Proceedings of the 21st National Information Systems Security Conference*, USA, pp. 303-314, 1998.

[MURR03] Murray, J., *An Exfiltration Subversion Demonstration*, Master's Thesis, Naval Postgraduate School, Monterey, CA, 2003.

[MYER80] Myers, P., *Subversion: The Neglected Aspect of Computer Security*, Master's Thesis, Naval Postgraduate School, Monterey, CA, 1980.

[NCSC95] Final Evaluation Report for the Gemini Trusted Network Processor, National Computer Security Center, Report No. 34-94, NCSC-FER-94/008, Ft. Meade, MD, 28 June 1995.

[OASD00] DoD Insider Threat Mitigation: Final Report of the Insider Threat Integrated Process Team, Office of the Assistant Secretary of Defense (C3I), April 2000.

[PARN72] Parnas, D.L., "On the Criteria To Be Used in Decomposing Systems into Modules," *Communications of the ACM*, v.15, n.12, pp. 1,053-1,058, December 1972.

[RAYM99] Raymond, Eric S., *The Cathedral & the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*, O'Reilly & Associates, October 1999.

[REED76] Reed, D.P., *Processor multiplexing in a layered operating system*, S.M thesis, Department of Electrical Engineering and Computer Science, MIT, Cambridge, MA, 1976. (Published as Technical Report MIT/LCS/TR-164, Lab. for Comp. Sci., MIT, Cambridge, MA, July 1976).

[REED79] Reed, D.P., and Kanodia, R.K., "Synchronization with eventcounts and sequencers," *Communications of the ACM*, v.22 n.2, pp.115-123, February 1979.

[ROGE03] Rogers, D., *A Framework for Dynamic Subversion*, Master's Thesis, Naval Postgraduate School, Monterey, CA, 2003.

[SCHE01] Schell, R.R., "Information security: science, pseudoscience, and flying pigs," *Proceedings of the 17th Annual Computer Security Applications Conference* (ACSAC 2001), pp. 205-216, 2001.

[SCHE03] Schell, Roger R., *private correspondence*, May 2003.

[SCHE85] Schell, Roger R., Tao, T.F., and Heckman, Mark, "Designing the GEMSOS security kernel for security and performance," *Proceedings of the 8th National Computer Security Conference*, pp. 108-119, 1985.

[SCHR77] Schroeder, M.D., Clark, D.D., and Saltzer, J.H., "The Multics Kernel Design Project," *Proceedings of the Sixth ACM Symposium on Operating Systems Principles*, pp. 43-56, November 1977.

[SEID90] Seiden, K., and Melanson, J., "The Auditing Facility for a VMM Security Kernel," *Proceedings of the IEEE Symposium on Security and Privacy*, IEEE Computer Society Press, pp. 262-277, 1990.

[THOM84] Thompson, K., "Reflections on Trusting Trust," *Communications of the ACM*, Vol. 27 No. 8, p. 761-763, August 1984.

[VIEG01] Viega, John, and McGraw, Gary, *Building Secure Software*, Addison-Wesley, 2001.

THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, VA

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, CA

3. Dr. Ernest McDuffie
   National Science Foundation
   Arlington, VA

4. David Ladd
   Microsoft Corporation
   Redmond, WA

5. Andy Allred
   Microsoft Corporation

6. Andy Newall
   Microsoft Corporation
   Redmond, WA

7. Jeana Jorgensen
   Microsoft Corporation
   Redmond, WA

8. Steve Lipner
   Microsoft Corporation
   Redmond, WA

9. Marcus Peinado
   Microsoft Corporation
   Redmond, WA

10. Marshall Potter
    Federal Aviation Administration
    Washington, DC

11. Ernest Lucier
    Federal Aviation Administration
    Washington, DC

12. Dr. Cynthia E. Irvine
Computer Science Department
Naval Postgraduate School
Monterey, CA

13. Dr. Roger Schell
Aesec Corporation
Pacific Grove, CA 93950

14. Lindsey Lack
Civilian, Naval Postgraduate School
Monterey, CA